**AFRL-RI-RS-TR-2009-161**
**Final Technical Report**
**June 2009**

# SELF-AWARE COMPUTING

Massachusetts Institute of Technology

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. AH09/00

***APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED*.**

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

**STINFO COPY**

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**
**ROME RESEARCH SITE**
**ROME, NEW YORK**

# NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-161 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/

CHRISTOPHER FLYNN
Work Unit Manager

/s/

EDWARD J. JONES, Acting Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

# REPORT DOCUMENTATION PAGE

*Form Approved*
**OMB No. 0704-0188**

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YYYY)*<br>JUN 09 | **2. REPORT TYPE**<br>Final | **3. DATES COVERED** *(From - To)*<br>Feb 07 – Nov 08 |

**4. TITLE AND SUBTITLE**

SELF-AWARE COMPUTING

**5a. CONTRACT NUMBER**
FA8750-07-1-0033

**5b. GRANT NUMBER**
N/A

**5c. PROGRAM ELEMENT NUMBER**
62303E

**6. AUTHOR(S)**

Anant Agarwal, Jason Miller, Jonathan Eastep, David Wentziaff and Harshad Kasture

**5d. PROJECT NUMBER**
AH09

**5e. TASK NUMBER**
MI

**5f. WORK UNIT NUMBER**
T1

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

Massachusetts Institute of Technology
77 Massachusetts Ave
Cambridge MA 02139-4301

**8. PERFORMING ORGANIZATION REPORT NUMBER**

N/A

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Defense Advanced Research Projects Agency
3701 N. Fairfax Dr.
Arlington VA 22203-1714

AFRL/RITB
525 Brooks Rd.
Rome NY 13441-4505

**10. SPONSOR/MONITOR'S ACRONYM(S)**
N/A

**11. SPONSORING/MONITORING AGENCY REPORT NUMBER**
AFRL-RI-RS-TR-2009-161

**12. DISTRIBUTION AVAILABILITY STATEMENT**
*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW 09-2755*

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

This project performed an initial exploration of a new concept for computer system design called Self-Aware Computing. A self-aware computer leverages a variety of hardware and software techniques to automatically adapt and optimize its behavior according to a set of high-level goals and its current environment. Self-aware computing systems are introspective, adaptive, self-healing, goal-oriented, and approximate. Because of these five key properties, they are efficient, resilient, and easy to program.

The self-aware design concept permeates all levels of a computing system including processor micro-architecture, operating systems, compilers, runtime systems, programming libraries, and applications. The maximum benefit is achieved when all of these layers are self-aware and can work together. However, self-aware concepts can be applied at any granularity to start making an impact today. This project investigated the use of self-aware concepts in the areas of micro-architecture, operating systems and programming libraries.

**15. SUBJECT TERMS**
Self-aware computing, self-optimization, fault tolerance

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON**<br>Christopher Flynn |
|---|---|---|---|---|---|
| **a. REPORT**<br>U | **b. ABSTRACT**<br>U | **c. THIS PAGE**<br>U | UU | 81 | **19b. TELEPHONE NUMBER** (*Include area code)*<br>N/A |

## Table of Contents

## List of Figures

## List of Tables

# 1 Introduction

Imagine a revolutionary computing chip that can observe its own execution and optimize its behavior around a user's or application's needs. Imagine a programming capability by which users can specify their desired goals rather than how to perform a task, along with constraints in terms of an energy budget, a time constraint, or simply a preference for an approximate answer over an exact answer. Imagine further a computing chip that performs better according to a user's preferred goal the longer it runs an application. Such an architecture will enable, for example, a handheld radio or a cell phone that can run cooler the longer the connection time. Or, a chip that can perform reliably and continuously in a range of environments by tolerating hard and transient failures through self healing.

This project proposes the vision of self-aware computation that will create such a computing device and an associated software system. A self-aware computing system is given a goal and a budget -- it then finds the best way to accomplish the goal with the means at hand. Much as in a biological organism, a self-aware (or organic) computer has five major properties:

1. It is INTROSPECTIVE or SELF-AWARE in that it can observe itself and optimize its behavior to meet its goals.

2. It is ADAPTIVE in that it observes the application behavior and adapts itself to optimize appropriate application metrics such as performance, power, or fault tolerance.

3. It is SELF HEALING in that it constantly monitors its resources for faults and takes corrective action as needed. Self healing can be viewed as an extremely important instance of self awareness and adaptivity.

4. It is GOAL ORIENTED in that it attempts to meet a user's or application's goals while optimizing constraints of interest.

5. It is APPROXIMATE in that it uses the least amount of precision to accomplish a given task. A self-aware computer can choose automatically between a range of representations to optimize execution -- from analog, to single bits to 64-bit words, to floating point, to multi-level logic.

# 2 Overview

Self-aware computation can be distinguished from existing computational models which are largely procedural. Today's models require a user to specify a procedure of how something is to be done and the computer blindly follows this procedure irrespective of application or environmental conditions using a fixed set of prearranged resources. For example, if the user wants to use a computing device for software radio, then the user programs it with a known bitwidth and prearranged code for algorithms such as Viterbi decode and Fast Fourier Transform (FFT) and to accept a given bitrate. The hardware is similarly fixed for all time. For example, the cache in the processing engine might be sized at 128 KB and two-way associative.

A self-aware computer, on the other hand, is given a goal and it attempts to achieve the goal with the minimal amount of resources and energy. Of course, it is also provided with many possible procedures to accomplish subtasks, each of which might use different types of architectural components. In our software radio example, the self-aware computer is given the goal of maintaining a connection to a receiver with a desired bit rate, using the least amount of energy. The software system and architecture collaborate on achieving this goal. A self-aware computer has cognitive hardware mechanisms in its trusted core to both OBSERVE and to AFFECT the execution. Since it is impossible to pre-configure all possible scenarios, the self-aware computer also implements learning and decision making engines in a judicious combination of hardware and software to determine the appropriate actions based on given observations. Thus, in our software radio example, the system will use the right precision for the FFT computations and the required amount of parallel hardware resources to achieve the goal. If the channel has very little noise, then the it might use a simpler coding scheme. The hardware will observe the execution of the code, and depending on the estimated working set size of the code, the system will shut off portions of the cache or make it direct mapped to save energy. At the same time, the system ensures that the goal is being met.

A self-aware computer can achieve 10x to 100x improvement in key metrics such as power efficiency and cost performance over extant computers. For instance, if for some streaming computation the system observes that 64 bits of precision is unnecessary (for example, if no changes are detected in the top 62 bits for a while) and can use 2 bits of precision, while at the same time turning off the data cache and using direct streaming of data over the network, the system can benefit from energy savings of 40x to 50x. As another example, the self-aware system might slow the clock to a sub module (and also the supply voltage) if its overall goal can be achieved with a much lower frequency. As a further example, in a tiled architecture running two streams of H.264 video encode, the system might observe the achieved output bandwidth for each stream, and move tiles between streams dynamically if the two video streams differ in complexity to maintain a fixed frame rate and a given per-stream bandwidth requirement.

Probably much more importantly, much like biological organisms, a self-aware computer can go well beyond traditional measures of goodness like performance and can adapt to different environments and even improve itself over time. It can also perform "code intrusion detection" by flagging abnormal behavior in its software by learning and maintaining signatures of its normal behavior. Corrective action might include shutting itself down or in some cases applying self healing. In doing so, the self-aware computer can build upon technologies developed for systems in the previous intrusion tolerant systems (ITS) program out of DARPA/IPTO in which a congruence between self healing for faults and for malicious intrusions was demonstrated.

Why now? Although such a machine may seem rather far fetched, we believe that basic semiconductor technology, computer architecture and software systems have advanced to the point that the time is ripe to realize such a system. To illustrate, let us examine each of the key aspects of self-aware computation including introspection, approximation, goal orientation, adaptation and self healing. We will discuss how they might be built in a practical way, and identify the fundamental challenges that we will have to overcome.

Introspection or self awareness implies that the system can observe itself while it is executing. The processor hardware can include mechanisms to observe instantaneous cache miss rates, bit positions in data words that are changing, cache sets that are hot versus others that are idle, numbers of errors in data transmissions or memory access, branch directions, network and memory latencies and queue lengths, among many others. These measurements will feed adaptation mechanisms that will adapt the architecture as needed. Introspection or self awareness requires foundational changes to computer architecture - self aware computers need mechanisms to observe themselves. Fortunately, semiconductor technology makes available billions of transistors on a single chip, so throwing transistors at the problem of building observers and recording state is eminently feasible today. Our challenge will be to identify what metrics are worth observing, how to make the measurements without impacting the execution, and what we can do with the results. For example, in recent unpublished work, we have shown that we can observe phase changes in program execution and change the cache access hash function to optimize cache miss rates. We have demonstrated that cache miss rates can be halved for many applications in this manner. In another body of work related to tiled architectures, we have used an adjacent helper tile to observe the execution (in particular, memory reference patterns) of a given master tile and prefetch data into the master tile's cache before it is needed [1].

Approximate computation implies that the computer does not always use the most available precision to accomplish a task. For instance, modern day processors have reached 64 bits of digital data widths. This data width is used in all computations whether it is needed or not. There are many classes of computations for which this precision is overkill. As an extreme example, imagine an image recognition task in which the final answer to the user is a single bit - yes or no (for example, is there a tank in this image or not). It is quite possible that a simple edge representation using just one bit per pixel might suffice to perform the pattern recognition task. Approximation can be applied in many other areas of digital design as well, and in fact, we question the very basic overkill digital design philosophy, *i.e.*, requiring signals to be fully restored after each logic element. The computer can try to use the minimal precision and probably even multilevel logic or the analog representation in its computations to produce a result. One research challenge with approximation is to figure out the minimal precision needed for a given computation. Another challenge will be to discover the best way to introduce analog representation into what is fundamentally a digital computer. Some recent work in this area includes compiler supported bitwidth analysis [2][3]. Other work that directly applies here is that of Rinard et al. [4][5] which has shown the possibility of highly reliable computation even when erroneous data values are ignored and allowed to propagate during program execution.

Goal orientation implies a revolutionary transition in architecture and algorithm design from a procedural style of specification to a goal oriented style. Goals indicate precisely what the user wants, not how to get there. This way, the computer can determine how best to achieve a user's goals depending on the conditions on the ground. Goal orientation can be applied at all levels of a system -- from the specification of the application all the way down to transmission of bits on a wire. In the latter case, a communications channel within the chip might choose to perform lossy compression to achieve effectively higher bandwidth transfer if the goal of the higher level application does not care about an exact representation. An example of an architectural goal can be to maintain no more than a maximum bandwidth demand on the memory system. An example of a system goal might be to maintain a given maximum power dissipation. Recent work along these lines includes the GOALS system [6][7][8] which is a software system that attempts to meet user-driven goals, rather than follow set procedures.

Adaptation is the ability of the computer to change what it is doing or how it is doing a given thing at run time. A key part of adaptation is the development of a control system as part of the computer architecture that observes execution, measures thresholds and compares them to goals, and then adapts the architecture or algorithms as needed. A key challenge is to identify what parts of a computer need to be adapted and to quantify the degree to which adaptation can afford savings in metrics of interest to us. Examples of mechanisms that can be adapted include various cache parameters such as associativity and replacement algorithm, prefetch methods, number of tiles used in a computation, and the presence or absence of coding or compression when transmitting data. Recent research on reactive synchronization [9][10] is another example of adaptation in which the waiting algorithm was tailored at run time to the observed delay in lock acquisition.

Self healing is an extremely important special case of adaptation. We give it independent billing because in the future era of multiple billions of transistors on a chip and deep submicron technologies, continuous correct operation in the presence of transient and hard failures will become a basic requirement. Thus a self healing system can use introspection to observe where errors are occurring and perform appropriate adaptation to fix the problem. For example, if errors are seen during data transmission on a given link, then the system can use one of two mechanisms to self heal. (1) It can use introduce coding to correct errors, or (2) it can cause messages to be rerouted to bypass the faulty region. The same technique can be used in caches to turn off cache banks that are producing errors.

Much like in the DARPA Polymorphic Computing Architectures program, self-aware computation applies to all levels of a computer system including computer architecture, VLSI chip design, operating systems, runtime software systems, compilers, programming libraries, and applications.

# 3 Summary of Accomplishments

The following is a summary of the major accomplishments achieved by this project. They will be described in detail later in the report.

1. Created a comprehensive vision for self-aware systems including key attributes, components, and interfaces. Prepared a detailed presentation describing this vision and offering many specific examples of ways that self-aware concepts can be incorporated into computing systems.

2. Designed the architecture for a self-aware *Organic Operating System* (OOS) including key components, interfaces, and required hardware support.

3. Developed the *Evolve* architecture and *Partner Cores* methodology to allow secondary cores on a multicore processor to observe and optimize the operation of a primary application core.

4. Created the *Organic Template Library* (OTL) which extends the C++ Standard Template Library with adaptive, self-optimizing data structures. These structures are able to dynamically adjust

various aspects of their underlying implementation including: the size of storage elements, the distribution of data across parallel nodes, and the internal organization of data (*e.g.,* list vs. tree).

5. Developed the concept of an *Organic Cache* which is composed of a single pool of memory that can be partitioned between data and instruction usage. Studied different algorithms for dynamically adjusting the partitioning during runtime and evaluated performance on several benchmarks.

6. Designed and began the implementation of *fos*, a "factored" operating system for large scale multicores. fos is designed from the ground up to be highly scalable, adaptable, and resilient by implementing OS services (*e.g.,* memory allocation, file systems, etc) as collections of cooperating servers distributed across a multicore chip. While fos is designed for a broad range of systems (not just self-aware systems), this project played a significant role in shaping some of its mechanisms and services.

7. Created *KLab*, a new large-scale, distributed multicore simulator based on Intel's Pin dynamic binary translator. This (partially complete) simulator allows us to simulate chips containing thousands of cores using a cluster of commodity workstations. The simulator uses flexible models of processing cores, caches, cache-coherence directories, on-chip networks, and DRAM that allow us to implement and experiment with new self-aware hardware components. For example, models of organic caches have been added to the simulator.

# 4 Accomplishments and Progress

This project identified and investigated several different techniques and mechanisms that can be used to build self-aware computer systems. They are described briefly in the following paragraphs. They can be applied individually to add self-aware capabilities to existing systems or all together to form a new type of self-aware computer system.

The heart of a complete self-aware system is a new "organic" operating system (OOS). The OOS acts as the ringleader, monitoring applications and making adjustments to hardware and software to increase performance, efficiency, and reliability automatically.

The Partner Cores framework provides mechanisms that allow secondary cores to monitor the execution of a primary core and perform on-line analysis and dynamic optimization of its application and hardware resources.

The Organic Template Library provides a set of data structures and algorithms (similar to the C++ Standard Template Library) that dynamically optimize their own implementations and behaviors. Because the OTL uses a standard sequential programming interface it is a good example of a self-aware technology that can be put use in today's existing systems.

An Organic Cache is a new hardware mechanism that allows a shared pool of local memory to be dynamically partitioned between different L1 caches and buffers. It is an example of a self-aware hardware component that would be managed by the OOS.

fos is a new type of operating system designed from the ground up for scalability, reliability and adaptability in large-scale multicore systems. It implements all system services using sets of servers distributed across multiple cores. fos provides an excellent framework on which to build a compete organic operation system.

KLab is a new distributed, parallel simulator for large-scale multicore processors. It provides an experimental testbench that can be used to study self-aware hardware and software ideas.

The following sections describe each of these ideas in more detail.

## 4.1 Organic Operating System

An Organic Operating System (OOS) is the heart of a complete self-aware system. It monitors application execution and hardware parameters and performs adjustments and optimizations to ensure that the applications are meeting their goals. There are four key components of an OOS: the application interface, the hardware interface, the analysis and optimization engine, and self-aware system services. Figure 1 shows how all of these components come together to form a self-aware system.

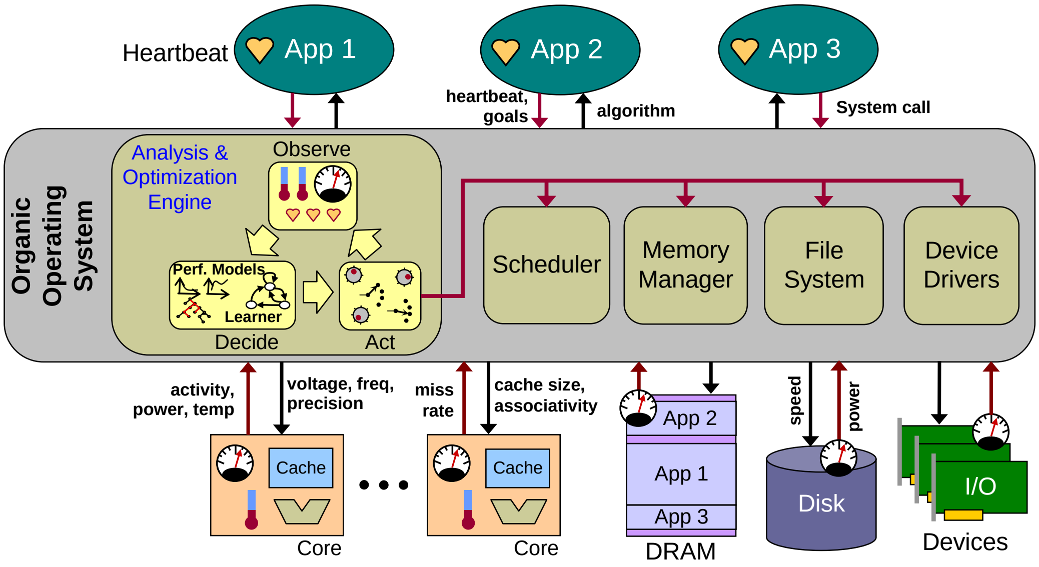


**Figure 1: Organic Operating System (OOS) architecture**

The application interface allows applications to communicate their goals and status to the OOS. Goals can relate to system characteristics (*e.g.*, minimize energy consumption) or application performance (*e.g.*, maintain a certain framerate). Application-specific performance information is delivered to the OOS using a *heartbeat*. A heartbeat is a periodic call that the application makes to the OOS to ensure that the application is still functioning and making progress. The OOS can use the intervals between heartbeats to measure application performance and verify that it is meeting its goals. For example, a video encoding application wishing to maintain a framerate of 30 fps, might make one heartbeat call for every frame encoded and specify a goal of maintaining 30 heartbeats per second.

The application interface is also used to control configurable parameters within an application. Many applications can be implemented with a variety of different algorithms or contain adjustable algorithmic parameters (*e.g.*, bounds on search or data blocking granularity). Often the programmer cannot be certain which algorithms or parameters will produce the best results at runtime. Rather than arbitrarily picking one, the programmer can implement multiple algorithms and allow the OOS to choose between them at runtime. In the video encoding example, the application could implement different algorithms that tradeoff image quality for performance. If the OOS detects that the application is not meeting its framerate goal, it could tell it to switch to a lower-quality but higher-performance algorithm.

Besides communicating with the application, the OOS needs to be able to monitor and control the system hardware. Future self-aware hardware platforms will contain a variety of sensors and counters to allow introspection. Individual components will maintain data about themselves and communicate it to the OOS. Examples include temperature, energy consumption, cache miss rates, and utilization. These components will also have control interfaces that allow the OOS to adjust things like hard drive spindle speed, frequency, voltage, and cache associativity. The design of new configurable hardware components is a fertile area for future self-aware research.

The analysis and optimization engine (AOE) is responsible for making optimization decisions based on the goals and data it receives from the applications and hardware. It employs a standard ODA (Observe, Decide, Act) loop to continually refine the system's operation and adapt to changing conditions. First, the ODA observes the current state of the system using the interfaces previously described. Next, it employs simplified models of component behavior and machine learning techniques to evaluate different potential optimizations and select the best options. Finally, the ODA acts by adjusting device configurations, changing application algorithms, or setting policies for system services. The ODA then observes the results of these changes and the cycle repeats. As the ODA tries different options, it updates its internal models and learns how to achieve optimal results.

The final component of an OOS is self-aware versions of standard system services such as file systems, schedulers, and memory managers. System service policies can have a large impact on overall application performance, particularly when multiple applications are running simultaneously. By varying those policies, the OOS can find a better overall result. For example, the OOS might instruct the scheduler to take time from an application that is running faster than needed and give it to one that is underperforming. If all apps are now exceeding their goals, the OOS might be able to lower the clock frequency and supply voltage to save energy.

Due to its unique ability to interact with both the hardware and software in a system, the operating system is able to take maximum advantage of self-aware concepts. The development of sophisticated organic operating systems, and particularly intelligent AOE's, is the key to the success of future self-aware systems.

## *4.2 Partner Cores*

Partner Cores [1] is an optimization framework for future computer systems with many cores. Its goal is to provide high performance, reliability, and low energy consumption through auto-tuning, insulating the programmer as much as possible from the added software complexity it would require to achieve this goal manually.

In the framework, some cores are devoted to running application code while others work as helpers or partners. Partners take as input hardware counter and other performance readings from application cores and run decision processes to optimize the execution of the application. Optimizations include performance tuning, fault tolerance, and minimizing energy consumption. As an example, a partner core can optimize an application's performance by examining its memory reference pattern, identifying a good prefetch algorithm for it, and then causing data to be prefetched into the application core's cache.

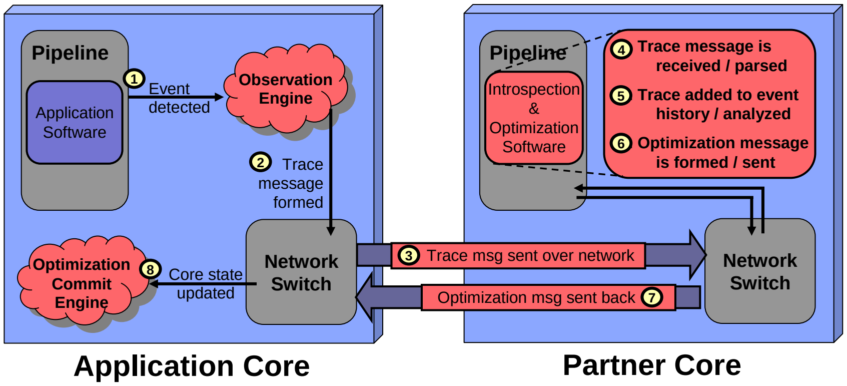

Pipeline
Application Software
1 Event detected
Observation Engine
2 Trace message formed
Network Switch
Optimization Commit Engine
8 Core state updated
Application Core
Pipeline
Introspection & Optimization Software
4 Trace message is received / parsed
5 Trace added to event history / analyzed
6 Optimization message is formed / sent
3 Trace msg sent over network
Optimization msg sent back 7
Network Switch
Partner Core

**Figure 2: Key operations in the Partner Cores framework**

The Partner Cores framework (shown in Figure 2) is a hardware and software framework. Application cores contain and expose hardware mechanisms for examining execution and monitoring various performance, reliability, and energy metrics. For example, one way of monitoring execution is to record traces of memory accesses. Examples for monitoring performance, reliability, and energy consumption include cache miss counters, execution signatures, and power readings, respectively. In addition to these observation mechanisms, application cores may contain special interfaces that allow a partner core to modify its state or cause events to happen in the background (such as prefetching a cache line). While hardware mechanisms are typically used on an application core, the analysis and optimization performed by a partner core is implemented in software. Typically, all cores would contain all of the required mechanisms so that cores can be flexibly assigned to either application or partner tasks.

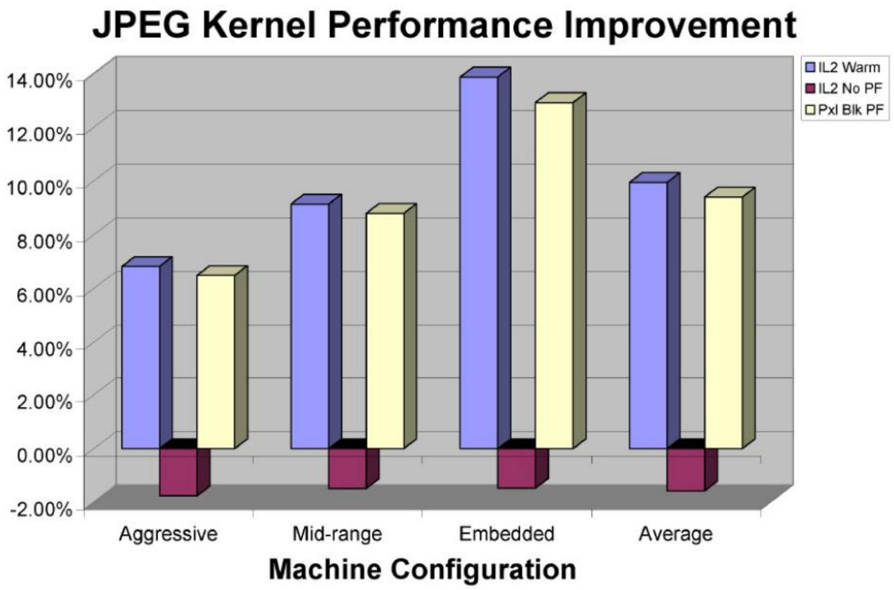


**Figure 3: Performance improvement of JPEG application using Partner Cores. Blue bars represent ideal performance, red bars the performance using basic software-caching, and yellow bars the performance with software-caching and intelligent partner-core prefetching.**

To evaluate the Partner Cores methodology, we extended the architecture of the Raw manycore processor [11] to include Partner Core mechanisms and implemented our changes in the Raw cycle-accurate simulator. As a case study, we implemented the partner core prefetching idea combined with a software data-caching scheme. Using a synthetic benchmark application modeled after JPEG image encoding, a partner core is used to implement a prefetching algorithm that is "aware" of the pixel block data structure used in the JPEG algorithm. As shown in Figure 3, for three different machine models representing a high-performance desktop, a mid-range desktop, and an embedded system, the partner cores approach achieves nearly optimal performance (as though all data were already in the cache).

The Partner Cores framework provides mechanisms for a wide variety of different optimizations and makes it possible to leverage the huge number of cores available on manycore processors to enhance existing sequential applications.

## *4.3 Organic Template Library*

We have developed a software library of common data structures and algorithms that programmers can use to make parallel programming easier. We call it the Organic Template Library (OTL). OTL applies bio-inspired adaptation strategies to auto-tune itself during program execution to optimize performance, parallelism, and power consumption so that programmers need not manually address these issues. OTL has been implemented for the Tilera TILE64 platform [12][13][14].

OTL is patterned after the ISO C++ Standard Template Library (STL) [15] in that OTL's interfaces for data structures and algorithms are similar. The OTL's interfaces present a sequential programming model but the underlying implementations are parallel. For suitable application domains, the sequential programming model provides a convenient abstraction that allows the programmer to ignore the complications of parallelism.

Under the hood, OTL's implementations are very different from STL's. OTL components dynamically self-optimize in response to runtime conditions and performance feedback, and they adapt to environmental factors such as input data characteristics and the availability of system resources. Some example observables analyzed during optimization are execution times, cache miss rates, memory load, network congestion, API usage history, and input data samples.

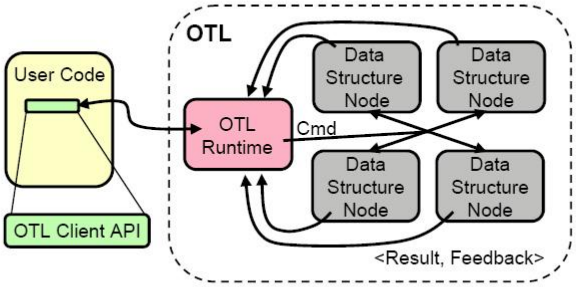
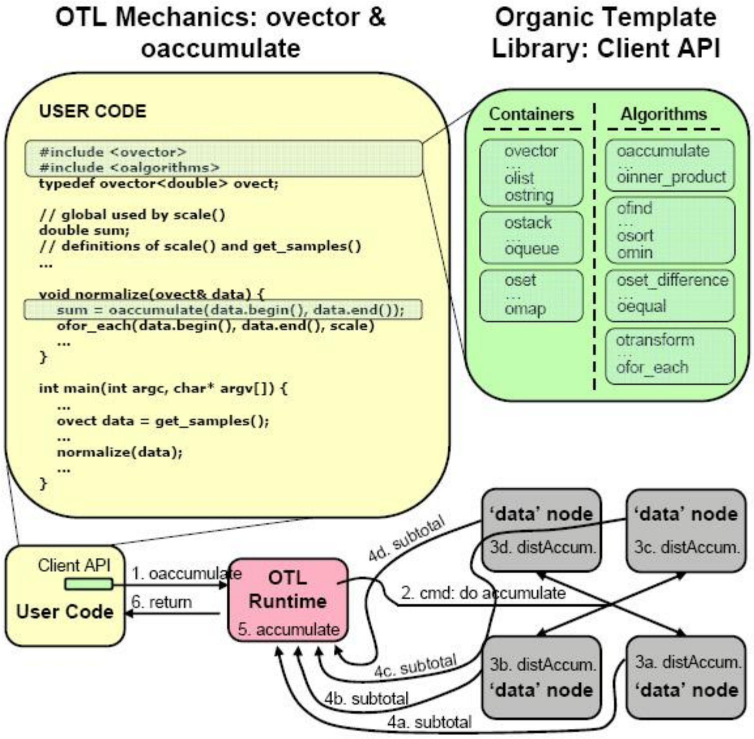


**Figure 4: Key components of the Organic Template Library**

The Organic Template Library consists of three parts: a client API, a server runtime system, and one or more data structure nodes (see Figure 4). The client API gets compiled into the user code. The runtime system and data structure nodes run alongside the user code concurrently on separate processors. The client API is the user code's interface to its OTL data structures. The client API shepherds user data structure method calls or algorithm calls to the runtime and responses from the runtime back to the user code. The runtime transparently manages the parallel workings and dynamic adaptation of the OTL. It translates user requests into commands which get issued to the data structure nodes. Taken together, the data structure nodes provide a distributed implementation of the OTL data structures and algorithms used by the user code. Object data is distributed across the nodes and the nodes contain code to carry out commands from the runtime.



```
#include <ovector>
#include <oalgorithms>
typedef ovector<double> ovect;

// global used by scale()
double sum;
// definitions of scale() and get_samples()
...

void normalize(ovect& data) {
   sum = oaccumulate(data.begin(), data.end());
   ofor_each(data.begin(), data.end(), scale)
   ...
}

int main(int argc, char* argv[]) {
   ...
   ovect data = get_samples();
   ...
   normalize(data);
   ...
}
```

**Figure 5: Invocation and mechanics of an OTL operation**

Figure 5 illustrates the mechanics of how a call to a method or algorithm operation in the client API gets executed in parallel by the runtime and data structure nodes. This example makes use of several organic containers and algorithms (*e.g.*, ovector, oaccumulate). These routines provide the same functionality and external interfaces as their standard STL counterparts (found by removing the leading "o" in the name) [15]. In the example, the user code instantiates an ovector, fills it with sample data, then normalizes that data. Normalization uses the oaccumulate and ofor_each algorithms. The example walks through how the OTL handles the first call, the oaccumulate call. First, the client API conveys what the user code wants to do to the runtime and waits for the action to complete. Then, the runtime looks up the ovector named "data" in a directory and sees that it is distributed across four data structure nodes. The runtime instructs each node to accumulate the data stored there and report the result back to the runtime. The data structure nodes do as instructed and then the runtime accumulates their subtotals and forwards the final total back to the client API. Finally, the oaccumulate call returns the value to the user code.

The runtime contains a runtime engine for each type of data structure supported in the OTL. Each runtime engine is responsible for controlling the adaptation of any live OTL objects of its type. The implementation of runtime engines varies somewhat from one data structure to the next since each data structure presents unique challenges and optimization opportunities. In general, however, a runtime engine draws from a specific basis of execution-time observables from which it is able to infer which choices it should make about algorithms, data organization, and parallelism. The runtime engine keeps a trace of data structure method and algorithm operation calls. It records performance statistics such as execution times, cache miss penalties, memory load, and network congestion by instrumenting the code in the data structure nodes. It samples object data (input characteristics) to discern its properties. Finally, the runtime monitors time-varying environmental factors such as a power budget or competition for resources among concurrently executing applications.

The trace of API calls gives information about the way data structures and algorithms are being used (*e.g.*, the relative frequencies of ofind and oinsert operations in a omap) which impacts what internal data organization should be used (a tree, list, or other). Performance feedback can be used to infer which algorithm is best in a specific situation (*e.g.*, which method of data partitioning and placement is highest performing for a particular hardware platform under a specific set of runtime conditions). Performance feedback can be combined with API call tracing to infer load at each processor and allow the runtime to redistribute computation. Input sampling can be used for algorithm adaptivity (*e.g.*, an ovector of data that is fairly pre-sorted already may be faster to sort using one method despite the fact that another method may be better in the general case). Finally, information about environmental factors enables the runtime to improve overall system performance or efficiency by adjusting the parallelism of its data structures.

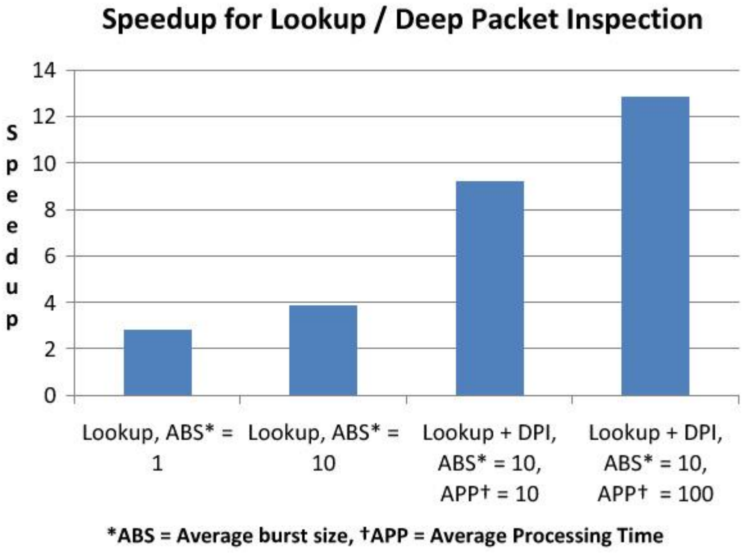


**Figure 6: Performance of Deep Packet Inspection application under various workloads**

To evaluate the OTL, we studied an important application in computer networking and security called deep packet inspection [16]. Deep packet inspection is a form of packet filtering that searches through the data and header of a packet, typically detecting and blocking things like intrusions, viruses, or spam or keeping statistical information for data mining. Using the OTL, we built a spam-blocker that searches packets for spam keywords and keeps statistics about packet sources to increase detection confidence. Statistics are associated with a packet identifier and stored in a map, an OTL omap. The omap API provides a callback that allows the programmer to remotely execute a kernel on a <key, value> pair. In this case, we use the pattern matching function as the callback. Internally, the omap intelligently load balances lookups and packet inspection computation across data structure nodes. As shown in Figure 6, OTL achieves up to 13x performance improvement over a baseline implementation across various input traffic patterns and levels of inspection detail. The baseline implementation is identical to the omap with optimizations turned off (*e.g.*, static data layout versus dynamic migration).

## 4.4 Organic Cache Memory

A typical modern processor (or core) contains several small memories including an instruction cache, a data cache, a TLB, and even a branch predictor. One of the tasks of a processor designer is to choose a size for each of memories, given a fixed total budget. However, different applications can place very different demands on these memories. For example, one application may execute a very small kernel of code that processes a huge amount of data, while another may execute a very long and complex algorithm on a small array. The first application needs very little storage for instructions but would benefit greatly from a large data cache. The second application has just the opposite needs. Therefore the fixed sizes chosen at design-time are always a compromise and are seldom ideal for any particular application.

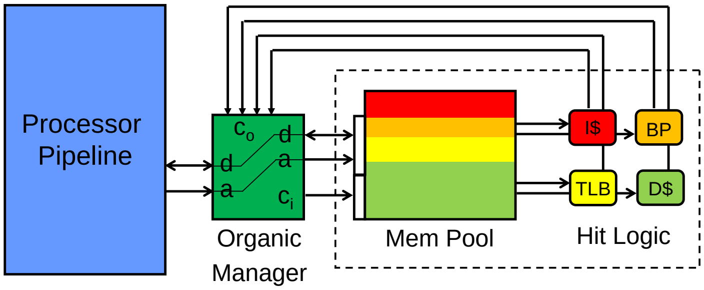


**Figure 7: Organic cache memory**

An “organic cache” is a single memory that can be dynamically partitioned among multiple uses as shown in Figure 7. It replaces all of the small memories in a processor with a flexible pool of memory that can be allocated to whichever use provides the most benefit at a given time. For example, a 32 KB organic cache could be configured to provide 16 KB of instruction cache and 16 KB of data cache or 4KB of instruction cache and 28 KB of data cache. This reconfiguration can be performed between applications or even dynamically within an application as it enters different phases of execution. An organic cache would allow a self-aware computer to automatically and transparently optimize the sizes of its caches and buffers to perfectly suit a program's needs.

To evaluate the benefits of an organic cache, we modified the Raw and KLab (see Section 4.6) simulators to include a cache that can be adjusted to trade-off instruction versus data capacity. The sizes of the two caches are adjusted by changing their associativity. For example, the organic cache can be configured as two 4-way set-associative caches or a 3-way cache plus a 5-way cache or a 2-way cache plus a 6-way cache, etc. Both caches always have the same number of lines so changing the number of ways in each line changes their total capacities.

**Table 1: Performance of an organic cache on the MPEG2 benchmark**

| Cache Configuration <I/D> | I$ misses | D$ misses | DRAM accesses | Savings over 32k/32k |
|---|---|---|---|---|
| **Static Configurations** | | | | |
| <32k/4-way, 32k/4-way> | 6658 | 414342 | 421000 | 0.00% |
| <24k/3-way, 40k/5-way> | 7815 | 392185 | 400000 | 4.99% |
| <16k/2-way, 48k/6-way> | 15949 | 385051 | 401000 | 4.75% |
| <8k/1-way, 56k/7-way> | 118417 | 382583 | 501000 | -19.00% |
| **Dynamic Configuration** | | | | |
| init: <32k/4-way, 32k/4-way> | 7747 | 391253 | 399000 | 5.23% |

Table 1 shows the benefits of an organic cache on the MPEG2 benchmark from the Mediabench suite [17]. The static configurations keep the division of resource constant during the entire program run. They essentially show the benefit of being able to customize the sizes of caches for a particular application. The dynamic configuration shows what happens when we allow the split between instruction and data caches to change dynamically at runtime. The caches are initialized to the 32k/32k configuration but the boundary is adjusted using a heuristic that pushes the I-cache smaller until the miss rate increases significantly. Even using this simple heuristic, the dynamic configuration outperforms all of the static configurations.

## 4.5 fos: A Factored Operating System

### 4.5.1 Overview

fos [18] is a new portable operating system designed from the ground up for scalability and targeted at 1000+ core systems. The main feature of fos is that each service that the OS provides is built like a distributed Internet server. Each service is implemented by multiple server processes which are spatially distributed across a multicore chip. These servers collaborate, exchange information and, in aggregate, provide the overall system service. fos distributes both high-level services as well as low-level services and data-structures typically found deep in OS kernels such as physical page allocation, scheduling, and memory management.

fos provides an excellent foundation for implementing new self-aware applications and OS services. The distributed nature is ideal for creating self-aware services that can observe and optimize an application as it runs. These services run on cores that are separate from the application, thereby allowing continuous monitoring of the application without stealing resources from it.

Implementing an OS kernel as a distributed set of servers has many advantages. First, internal OS communication is made explicit and exposed, thus making it easier to troubleshoot and optimize. Second, the number of servers can be varied based on the number of cores or other system characteristics, providing scalability and the ability to adapt to changing conditions or demands. Third, because servers run on dedicated cores, the operating system and application do not compete for local resources such as caches and TLBs. Finally, because there are multiple servers for each service, there are no single points of failure in the system. If an OS or user core has a failure, one of several introspection cores will observe the problem and cause the affected server or application code to be restarted on a different core.

### 4.5.2 Architecture

A factored operating system environment is composed of three main components: a thin hypervisor, sets of servers that together provide system services (which we call the OS layer), and applications that utilize these services. The lowest level of software management comes from the hypervisor. A portion of the hypervisor executes on each processor core to control access to resources (protection) and provide a core-to-core communication API. Applications and servers execute on top of the hypervisor and share core resources with the hypervisor.

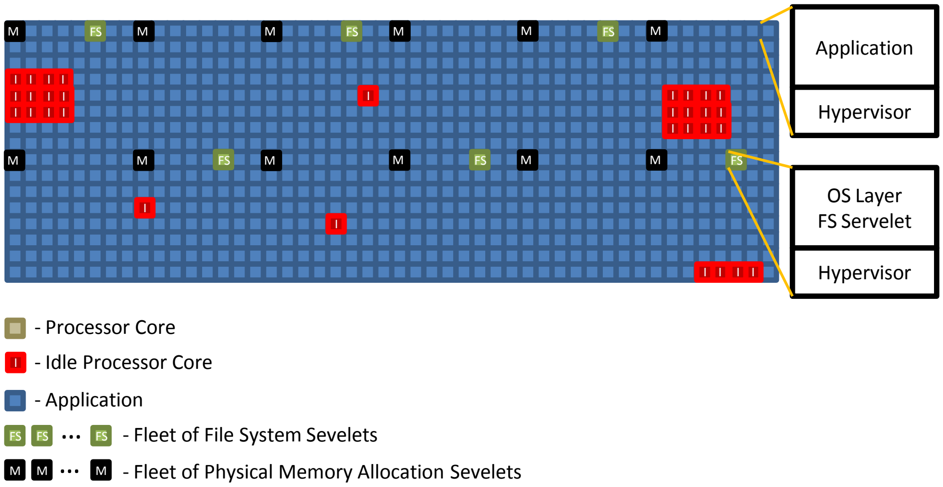


**Figure 8: Example of fos OS and application clients executing on a large multicore processor**

The OS layer is composed of sets of function-specific servers. Each core operating system function is provided by a different set made up of one or more servers. For instance, there is a set that manages physical memory allocation, a set that manages file system access, and a set that manages process scheduling and layout. As shown in Figure 8, the servers within a set are distributed across the multicore chip to provide local access for the cores in their areas. By default each server executes solely on a dedicated processor core. Servers communicate only via the messaging interface provided by the hypervisor layer.

In fos, an application executes on one or more cores. Within an application, cores may communicate using either shared memory or messaging, depending on what the hardware supports. The OS layer uses only explicit messages (which can be implemented efficiently on both shared-memory and message-passing machines) for internal communication. When an application requires OS services, the underlying communication mechanism is via hypervisor messaging. A more traditional system-call interface is exposed to the application writer and a small translation library is used to turn system calls into messages to an appropriate server.

Currently a basic fos hypervisor has been implemented. It contains a messaging API with messaging layer allowing servers to communicate. A spawning and memory management API has been developed and is being added into the fos hypervisor. A proof-of-concept system service has been built and runs within fos. More advanced system services are currently being developed. fos is currently running on x86_64 hardware and emulators, but is designed to be easily ported to other architectures.

When ready, fos will be released to the open source community to foster additional research on self-aware multicore systems.

fos will be the basis for a new self-aware operating system. System servers will be developed that can detect the run-time needs of the application mix and spawn new servers to meet application requirements. By changing the number of servers dedicated to each service, fos is able to scale itself up or down as needed. By building the appropriate interfaces into these services, fos will be able to optimize itself and the applications it is running to meet their runtime goals and requirements.

## *4.6 KLab Multicore Simulator*

To experiment with new self-aware hardware mechanisms and massive multicore processors that do not yet physically exist, we have developed the KLab multicore simulator. KLab is a fast, flexible simulator designed to simulate future large-scale multicore processors on today's multicore servers. To achieve the performance necessary to effectively simulate such large systems, KLab was designed from outset to run in parallel across a cluster of servers.

KLab is implemented using Intel's Pin dynamic binary instrumentation infrastructure [19]. Pin allows one to modify an application as it is running. Using Pin, we can allow the majority of the application being studied to execute directly on the host machine for maximum performance. Where there are differences between the host machine and the simulated machine, Pin is used to insert code to simulate the new experimental features. Pin can also insert code into the program to model the performance of the application on the simulated system.

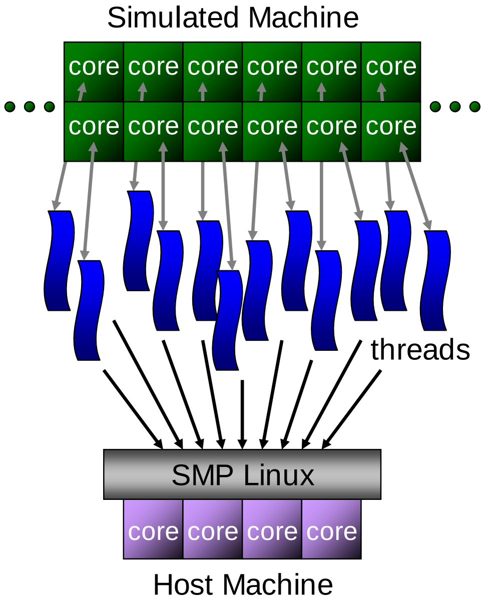


**Figure 9: Core-to-thread mapping in the KLab simulator**

As shown in Figure 9, KLab has been designed to take advantage of the parallelism of host architectures such as multicores or clusters. It models each core within the simulated system using a separate kernel thread, independently schedulable by the OS. The OS maps the threads to the host hardware, enabling the simulator to exploit the available parallelism. Cores (threads) communicate using calls to a simple API which represents the intrinsic capabilities of the simulated architecture, *e.g.*, broadcast, point-to-point message-passing, etc. KLab replaces API calls within the application with calls to simulator functions that implement the corresponding functionality and update the simulation clock of the appropriate cores using a model of the communication cost. The implementation of the API functions within the simulator depends on the communication mechanisms available on the host architecture. For example, the implementation of inter-core communication uses buffers in shared memory for threads on the same machine and MPI over Ethernet for threads running on different machines in a cluster.

We have chosen to base our multicore simulator on Pin because it offers several advantages over creating our own simulator from scratch. First, the Pin infrastructure is reliable: it is mature, robust, and well-supported. Second, it is high performance: it natively executes application code on the host hardware rather than interpreting it. Third, using Pin shortens our simulator toolchain development time: it allows us to use existing tools for compiling multicore applications (gcc, binutils, etc.) instead of having to develop them ourselves.

# 5 Conclusions and Recommendations for Future Work

This project has defined the key components of a self-aware computer system and investigated several promising examples of those components. We believe that the rapidly growing complexity of modern machines combined with the massive quantities of computational resources those machines provide has created the perfect setting for self-aware systems. Whereas existing systems are designed to simply follow instructions, future systems must be intelligent to help the programmer deal with their baffling complexity. Happily, that complexity also means that there are plenty of resources available (in the form of cores or transistors) to implement that intelligence. Self-aware systems are the key to maximizing performance and efficiency while increasing programmer productivity.

The field of self-aware computing is in its infancy and therefore there is plenty of room for future research. The most important (and challenging) goal is the development of intelligent algorithms for making optimization decisions. Fortunately there is already a large body of work in machine learning that can be applied to these problems [20][21][22].

However, before we can make optimization decisions, we need to have quality information. Therefore the development of good APIs for communicating goals, capabilities and status is also crucial. In particular, the concept of an application heartbeat is a simple but powerful of idea that warrants additional study. Where should heartbeat calls be inserted into a program? What information should be passed to the OS with each heartbeat? How should goals be specified in terms of heartbeats?

Even *perfect* optimization decisions are useless without parameters to tune. Additional research is needed in flexible, configurable hardware and software components that will give the OOS something to adjust. We believe that some of the enormous quantities of transistors that will be available in future chips should be applied to make systems easier to use instead of simply providing raw resources (such as additional cores). Hardware components that can monitor and report on their own behavior and morph themselves into different modes will make self-aware systems much more efficient.

# 6 References

[1] Jonathan Eastep. *Evolve: A preliminary multicore architecture for introspective computing.* Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2007.

[2] Mark Stephenson, Johnathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, Jun 2000.

[3] Altaf Abdul Gaffar, Oskar Mencer, Wayne Luk, Peter Y.K. Cheung, and Nabeel Shirazi. Floating-point bitwidth analysis via automatic differentiation. In *FPT'02: Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 158–165, Dec 2002.

[4] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2004.

[5] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS '06: Proceedings of the 20th annual International Conference on Supercomputing*, pages 324–334, New York, NY, USA, 2006.

[6] Umar Saif Hubert, Hubert Pham, Justin Mazzola Paluska, Jason Waterman, Chris Terman, and Steve Ward. A case for goal-oriented programming semantics. In *System Support for Ubiquitous Computing Workshop at the Fifth Annual Conference on Ubiquitous Computing*, 2003.

[7] Justin Mazzola Paluska. *Automatic implementation generation for pervasive applications.* Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2004.

[8] Justin Mazzola Paluska, Hubert Pham, Umar Saif, Chris Terman, and Steve Ward. Reducing configuration overhead with goal-oriented programming. In *PERCOMW'06: Proceedings of the 4th Annual IEEE International Conference on Pervasive Computing and Communication Workshops*, March 2006.

[9] Beng-Hong Lim and Anant Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *ASPLOS VI: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, San Jose, CA, October 1994.

[10] Beng-Hong Lim. *Reactive synchronization algorithms for multiprocessors.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[11] Michael Bedford Taylor, Jason Kim, Jason Eric Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, Mar 2002.

[12] Max Baron. Tilera's cores communicate better. *Microprocessor Report*, November 2007.

[13] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V . Leung, J. MacKay, and M. Reif. TILE64 Processor: A 64-Core SoC with Mesh Interconnect. In *International Solid-State Circuits Conference*, 2008.

[14] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, Sept-Oct 2007.

[15] Nicolai M. Josuttis. *The C++ Standard Library : A Tutorial and Reference*. Addison-Wesley Professional, August 1999.

[16] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–350, New York, NY, USA, 2006.

[17] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO-30: Proceedings of the 30th annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[18] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review, Special Issue on the Interaction among the OS, Compilers, and Multicore Processors*, Apr 2009.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005.

[20] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, March 1997.

[21] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, March 1998.

[22] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, March 2000.

# 7 List of Symbols, Abbreviations and Acronyms

AOE – Analysis and Optimization Engine

API – Application Programming Interface

D$ – Data cache

DARPA – Defense Advanced Research Projects Agency

DRAM – Dynamic Random-Access Memory

FFT – Fast Fourier Transform

fps – Frames Per Second

I$ – Instruction cache

IPTO – Information Processing Techniques Office (division of DARPA)

ISO – International Standards Organization

ITS – Intrusion Tolerant Systems

KB – Kilobyte

MPI – Message Passing Interface

ODA – Observe, Decide, Act

OOS – Organic Operating System

OS – Operating System

OTL – Organic Template Library

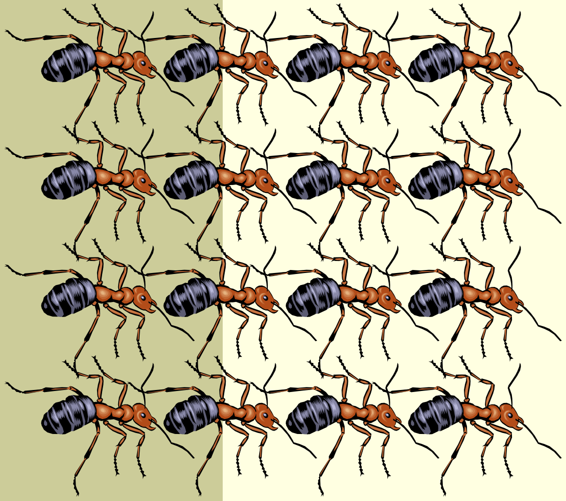
STL – Standard Template Library

TLB – Translation Lookaside Buffer

VLSI – Very Large Scale Integration

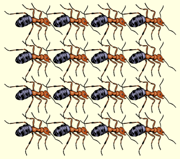
# Appendix 1

## Self Aware Computing Briefing



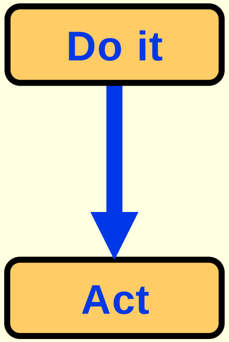
# Self-Aware Computing

Anant Agarwal
MIT CSAIL

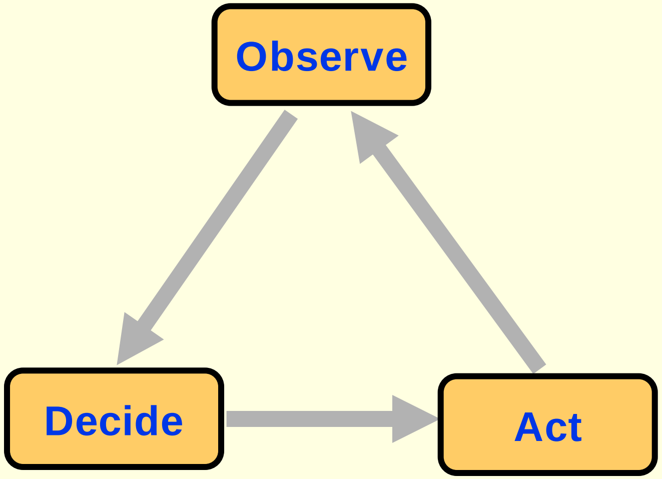


# Characteristics of Self-Aware Systems

- Current systems are procedural
  - Their behavior is pre-programmed
  - Based on guesses about resource availability
  - Ill-suited to complex multicore systems
  - Results in sub-optimal performance in the face of changing conditions



- Self aware systems learn how they can be used to address a particular problem
  - Respond to user goals
  - Build self-performance models
  - Identify what they needs to learn
  - Adapt to changing goals, resources, models, operating conditions
  - Gracefully adapt to failures
  - Optimize their own behavior



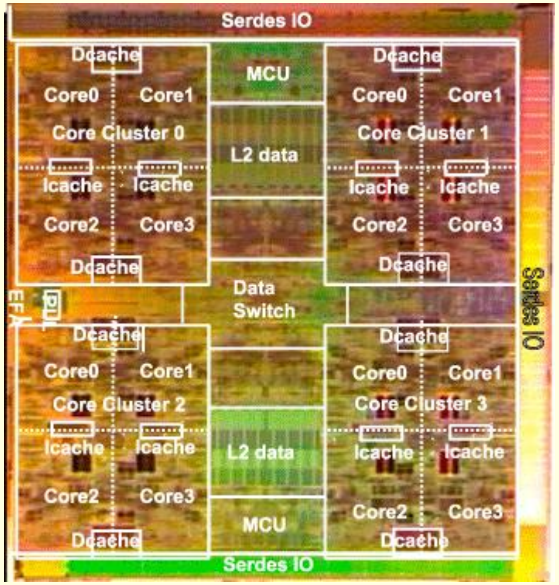
# Outline

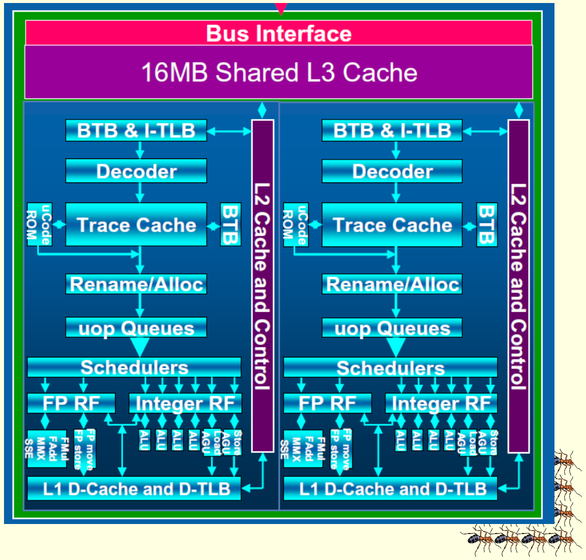
- Introduction
- Vision and Goals
- Limitations of Today's Systems
- How to Build a Self-Aware System
- Why Now?
- Potential Impact
- Program Timeline

# System Complexity Becoming Unmanageable

- System complexity is increasing rapidly
  - Multicore architectures with parallelism and heterogeneity at many levels
  - Distributed, deep, and heterogeneous memory hierarchies
  - Special-purpose functional units and special instructions
  - Unreliable components (hard and soft errors)
  - Physics introducing new constraints such as power, energy and wire delay

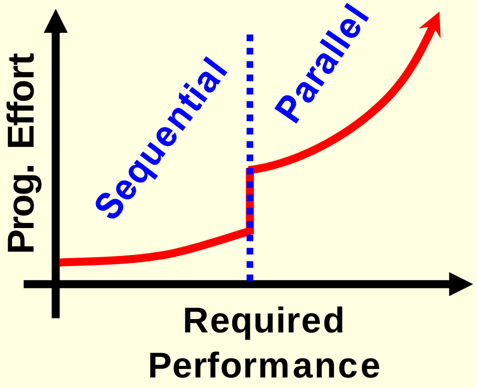
- Traditional abstraction layers are failing
  - The ultra-wide sequential superscalar processor is dead. Multicore and parallelism are the future
  - Additional complexity is dumped on programmers' shoulders
  - Complete system models are nuance-ridden and too complex to comprehend
  - Programmers cannot make efficient use of the available resources
  - Too many possible failure modes to anticipate
  - Current abstractions and APIs do not comprehend the new constraints
  - Static and brittle designs – currently developer must anticipate all mission and system dynamic changes

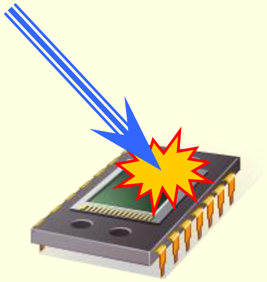




# The Consequences

- Programming has become very difficult
  - Our programming models are in the dark ages
  - Parallel programming requires experts
  - Impossible to balance all constraints manually
- Suboptimal results
  - Systems are too complex for programmers to understand
  - Programmers have no idea how to optimize energy utilization
- No program portability
  - Impossible to write programs that perform well on a large variety of machines
- Failure rates are increasing
  - Smaller devices more susceptible to cosmic rays, manufacturing variations, electromigration, thermal variations
  - With huge numbers of devices, even low-probability events happen frequently
  - Anecdotal evidence that today's servers fail routinely – software often blamed
- Development costs are skyrocketing
  - Verification and validation is increasingly challenging
  - Code development and optimization taking a lot longer
  - The n-squared problem: Each new environment needs independent validation and optimization
  - Several validation engineers to a single developer



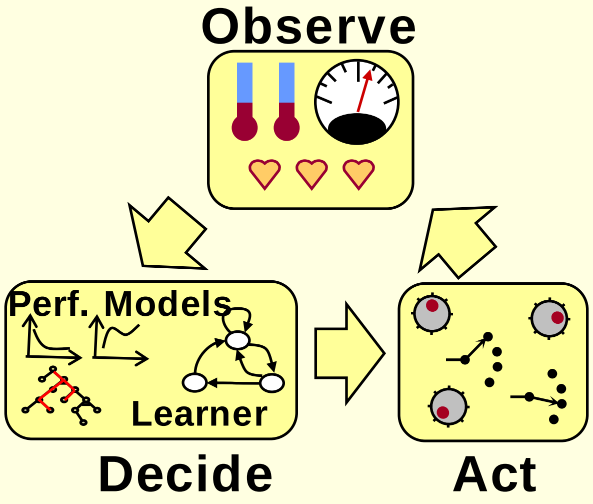


# Outline

- Introduction
- Vision and Goals
- Limitations of Today’s Systems
- How to Build a Self-Aware System
- Why Now?
- Potential Impact
- Program Timeline

# The Vision: Self-Aware Computing

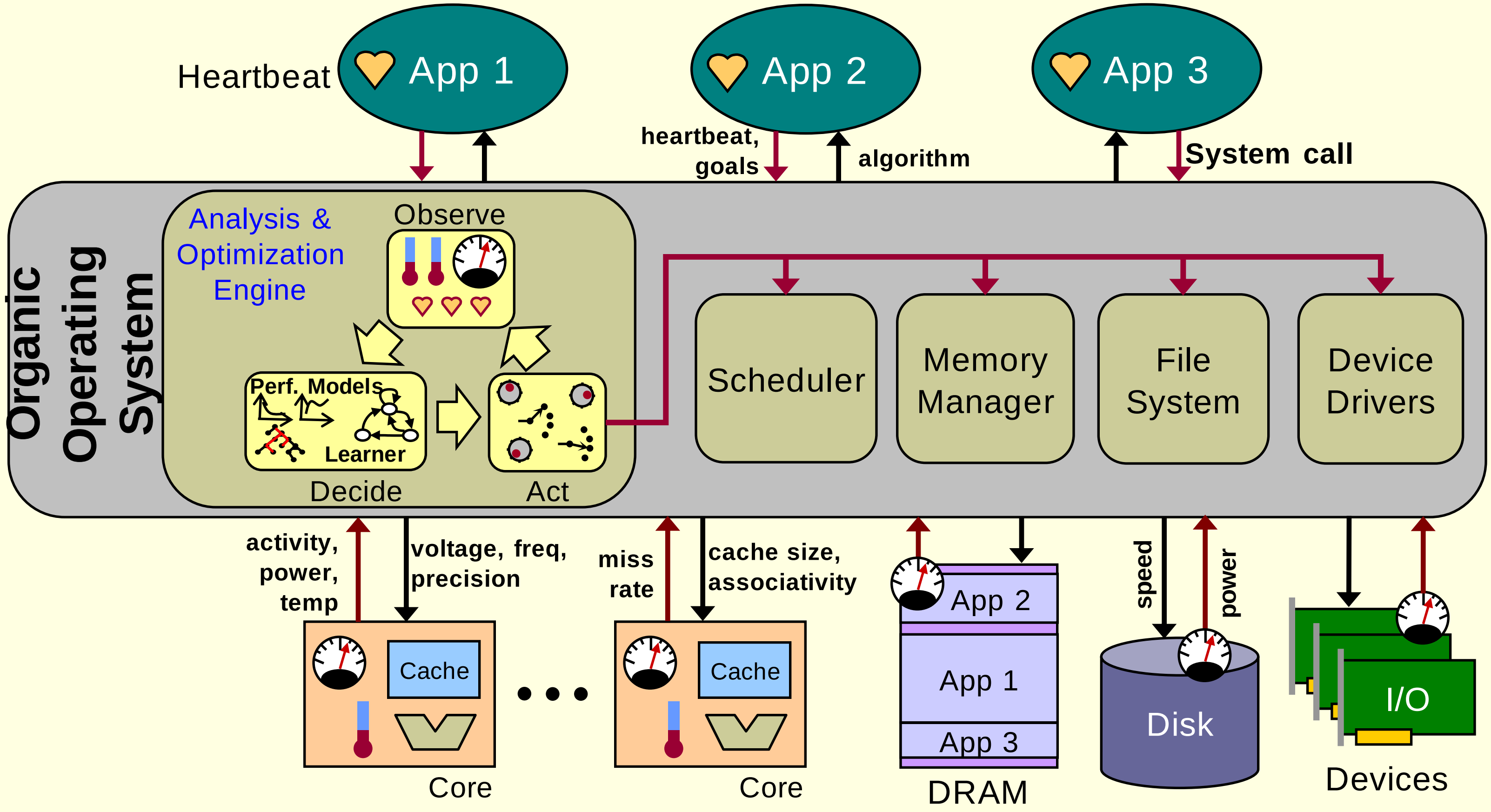
Self-Aware Computing – a computing paradigm characterized by systems that can observe their runtime behavior, learn, and take actions to meet desired goals



A self-aware system is

- Introspective
  - Observes itself, reflects on its behavior and learns
- Goal-oriented
  - Tell computer what you want, computer's job to figure out how to get there
- Adaptive
  - System analyzes the observations, computing the delta between the goal and observed state, and takes actions to optimize its behavior
- Self-healing
  - System continues to function through faults and degrades gracefully
- Approximate
  - System does not expend any more effort than necessary to meet goals

# A Self-Aware Computing System



- An Organic Operating System (OOS) is a key enabling technology for self-aware systems
- OOS includes learning based ODA loop to optimize resource management
- Observation and control interfaces added to all apps, SW and HW components.
  - Observe temp, heartbeat/performance, miss rates, queue lengths, util. of resources, etc.
  - Control alg., #cores allocated, cache config, scheduler policy, affinity, freq., precision
- Application communicates goals and options to OOS
- OOS uses component perf. models to decide how best to meet goals under given system constraints (e.g. performance, quality, temp, power)

# The Possibilities…

- Imagine a 1K-multicore serving up a computational video application that runs cooler and produces higher quality video the longer it runs

- Imagine a piece of code that can run on a massive multicore server producing high-quality results while meeting a real-time goal …

  … and can also run on a 4-core handheld radio meeting the same real-time goal, but compromising somewhat on result quality

# Self-Aware Computing Goals

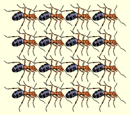
- To build general purpose systems that can meet targets (such as performance, reliability, power) while satisfying certain constraints (power, energy, area) under changing mission conditions and dynamic ground situations

- To build easy-to-program systems where the user does not have to understand the interaction between system components and write code for every specific combination of conditions, and where the programmer does not have to maintain a complete system model in their mind

- To build portable systems where the user does not have to manually redesign, rewrite, and retune code for each new system or environment, where the system automatically optimizes for different platforms

- To build systems that are automatically resilient to faults

Rethinking computer systems to reflect 21st century constraints and opportunities



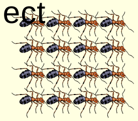
# Self-Aware Computing Program

- Although creation of complete self-aware systems is the ultimate goal, this program takes the first steps by developing the key enabling technologies including an Organic OS and minimum set of related components and APIs. Specifically, this program will create:
  - 1. Self aware operating system – rethink operating systems from the ground up
  - 2. Software and hardware components minimally enabled for operation within a self-aware system
  - 3. APIs for self-aware interactions

- 1. Develop brand new self-aware operating system: OOS
  - Experimental, research-grade implementation
  - Complete Observation, Decision making, and Action engine (ODA loop)
  - Evaluation of various competing technologies for learning and decision making

- 2. Develop performance models and prototypes of software and hardware components required to support OOS, for example:
  - Observable and controllable schedulers and device drivers
  - Applications that establish goals and targets for performance and reliability
  - Adaptive, introspective data structures
  - Reconfigurable caches that report miss rates, utilization, etc.
  - Adaptive I/O (adjustable voltage/frequency)

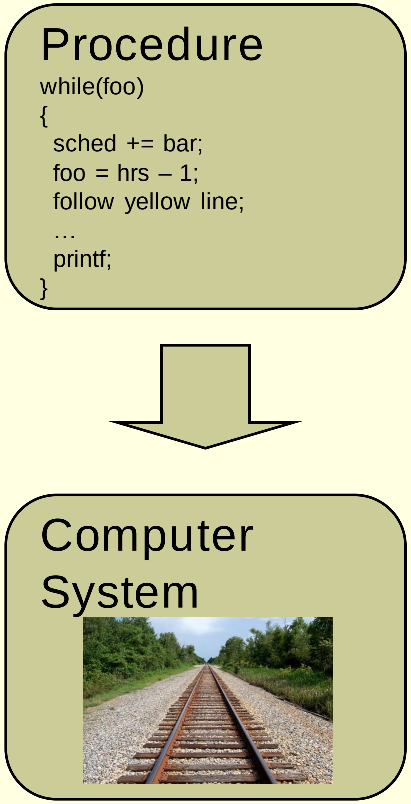
- 3. Develop APIs for communication with components and applications
  - For applications to communicate their goals, configurable parameters, and internal component models to the OOS
  - For OOS to receive status information from software and hardware components and control their configurations
  - How to access and organize the massive amounts of data that the software and hardware can collect.



# Outline

- Introduction
- Vision and Goals
- Limitations of Today’s Systems
- How to Build a Self-Aware System
- Why Now?
- Potential Impact
- Program Timeline

# Present Day



Procedure

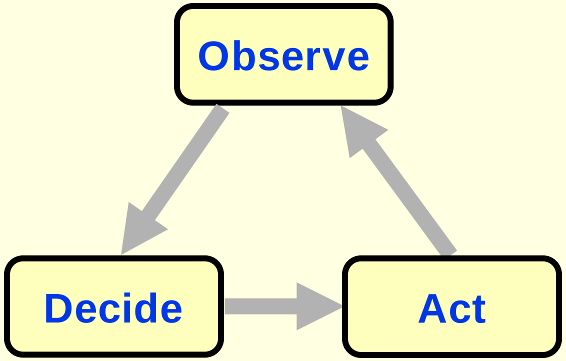
```
while(foo)
{
  sched += bar;
  foo = hrs – 1;
  follow yellow line;
  …
  printf;
}
```

Computer System

- With some exceptions, mostly procedural approach to all aspects of computing
- Behavior is completely specified ahead-of-time by programmer or system developer
- User manually applies goals and constraints to each individual component, not to system as a whole
- This problem applies to architecture, runtimes, compilers, operating systems, languages

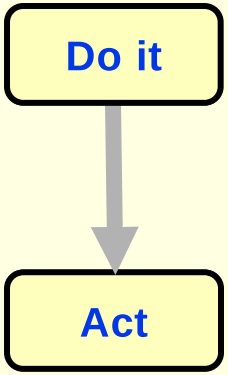
# A Contrast – Self Aware vs Procedural

## Self Aware



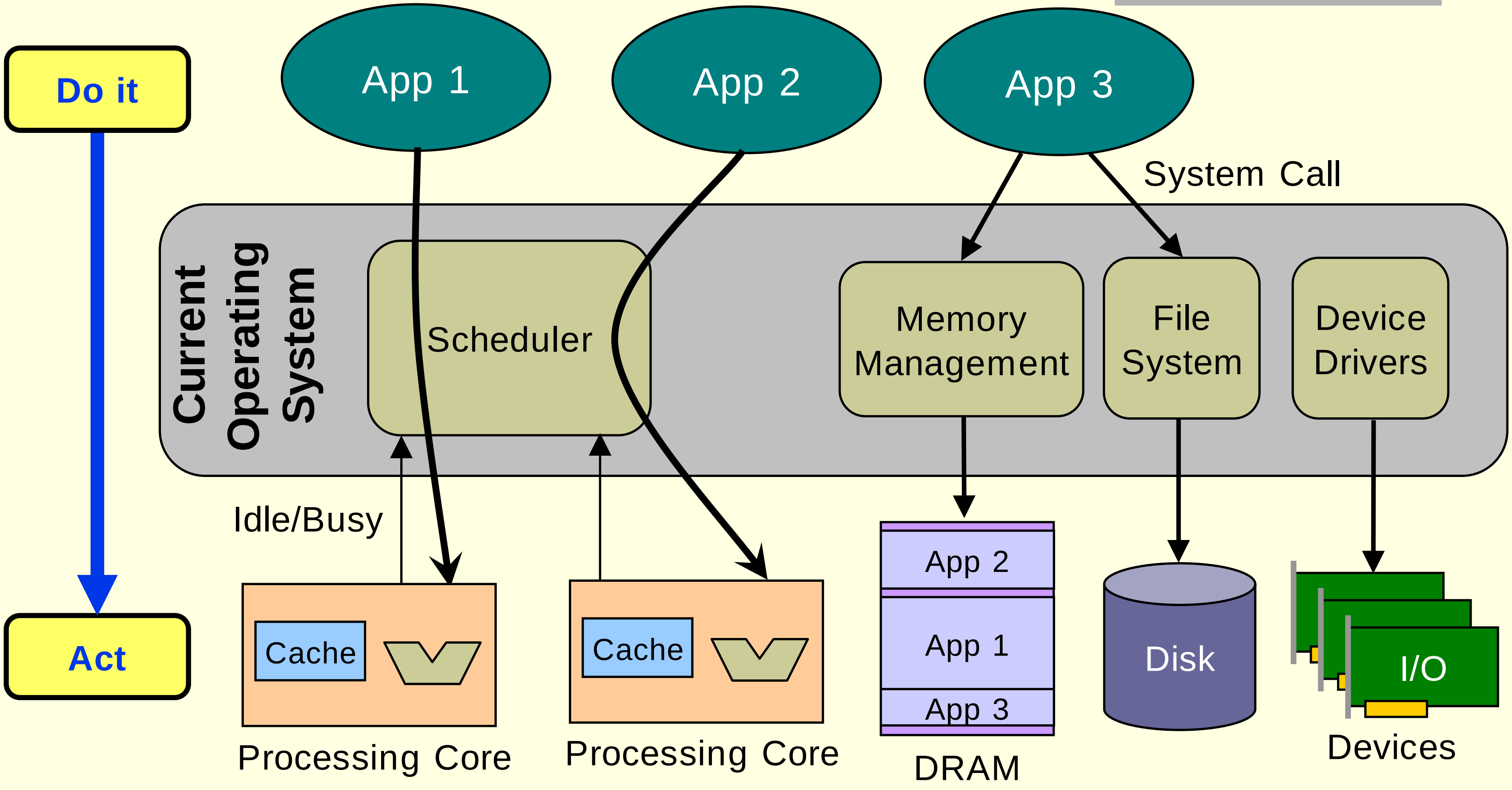
Self-aware frameworks show the characteristic ODA loop

## Procedural



Procedural frameworks run open loop

# Current Operating Systems



- Procedural: Responds to immediate requests (via system calls) from apps
- Runs open loop: No feedback from hardware/software (inferred from requests)
- No long-term goals or performance model to guide decisions

# What's Wrong With This

- No automatic adjustment to conditions
  - Computer blindly follows instructions
  - Programmer is responsible for any dynamic behavior
  - Programmer must anticipate all possible operating conditions
    - Different quantities of resources
    - Different energy envelopes
    - Unexpected input/mission threats
  - Poor code portability
- Not fault tolerant
  - Programming model assumes all parts work perfectly at all times
  - Impossible for programmer to consider all possible failure modes
  - Whole-system checkpointing with rollback is not practical for real-time systems or frequent errors
- Non optimal
  - Individual component optimization is non optimal
  - Need to overprovision resources in all dimensions
  - User must manually compose constraints into a global end to end number

# The Result?

- Programming difficulty and effort is exploding
  - Systems becoming too complex to internalize
  - All this complexity is pushed to the programmer
  - Programmer is responsible for handling all possible conditions/failures
  - Very difficult to understand the interaction of all constraints
- Systems are expensive and fragile
  - Overprovisioning of resources in all dimensions results in expensive systems
  - Even slight variations in deployment conditions result in unpredictable behavior and errors
- In reality, programmers cannot handle this today
  - Unhandled exceptions, latent bugs
  - Programs must be rewritten/reoptimized to run on different machines
  - Most applications are not fault-tolerant

# A Remote Briefing Example

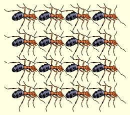
Situation Room Telepresence

Soldier With Handheld

- Situation
  - Imagine a videoconferencing system painstakingly tuned for highest quality (e.g., 1080p resolution) for remote briefings between situation rooms using networking links with guaranteed QOS
  - Suppose we want to port the system for briefings to soldiers in the field using energy-efficient handhelds with lower quality (e.g., QCIF resolution)
- The present day
  - First, porting the software is a huge effort
  - Second, even with an expensive port, it will drain the power of the handheld in minutes
  - Further, video is unusable, because it is jumpy due to uneven frame rate caused by variable link conditions
- Self aware systems
  - A self-aware system, on the other hand, would allow us to run the same application on the handheld without a porting effort
  - It would adapt to the new screen resolution automatically, lower video quality and meet an energy constraint using energy optimized algorithms, and maintain an even framerate using interpolation

# A Supercomputing Example

- The situation
  - Imagine a supercomputer built out of a large number of components
  - Suppose it must perform a long running computation
- The present day
  - The programmer must program in resilience (*i.e.,* manually code to detect there is an error, for example due to a wire break due to metal migration in some component)
  - The programmer must introduce code for checkpointing, error detection and rollback for each application. This approach is static, complicated, and error prone
  - Because it is static, system performance degrades significantly as the faulty component causes frequent errors and rollbacks. The system may also require a manual diagnostic check to remove the faulty component
- Self aware systems
  - Self aware system is automatically resilient to faults and requires no manual checkpointing
  - It will dynamically localize the error through self checks, and route around the faulty link

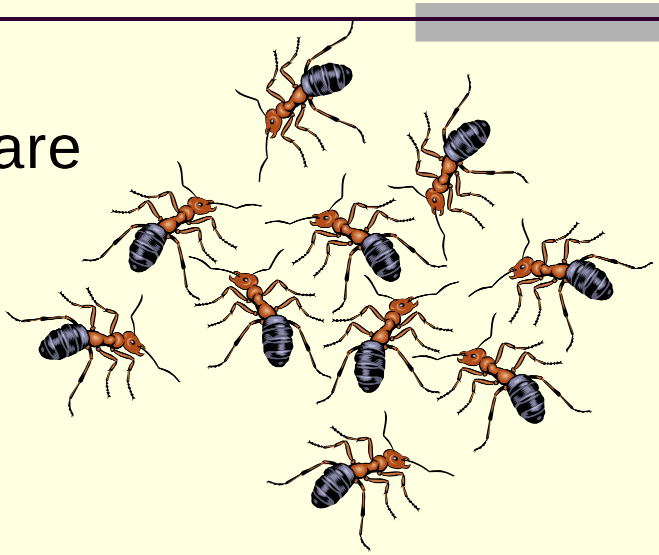


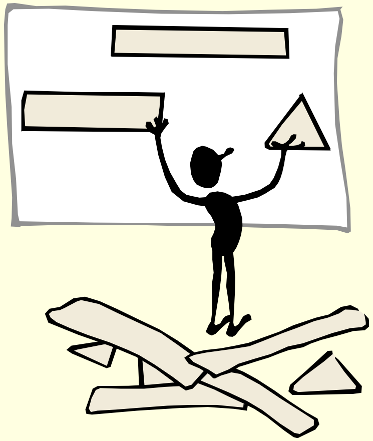
# Outline

- Introduction
- Vision and Goals
- Limitations of Today’s Systems
- How to Build a Self-Aware System
- Why Now?
- Potential Impact
- Program Timeline

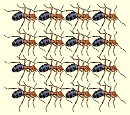
# The Inspiration

- Nature abounds in complex, highly-parallel systems that are
  - Adaptive
  - Self-healing
  - Evolving
  - Goal-oriented

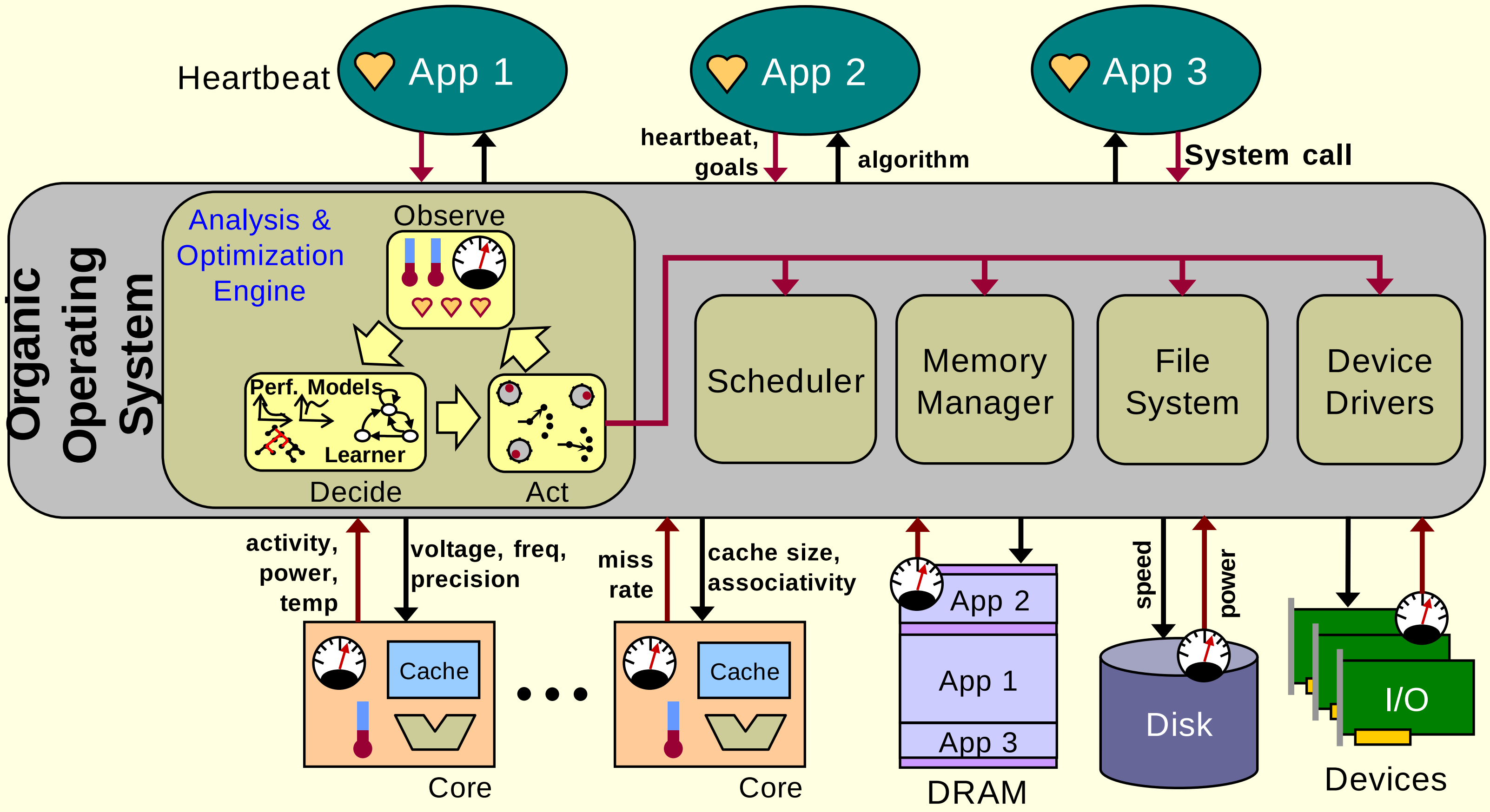




- … while our engineering disciplines approach systems using
  - Predetermined algorithms
  - Manual procedural programming
  - Deterministic (understandable) global state models
  - Fragile assumptions (“all parts work”)

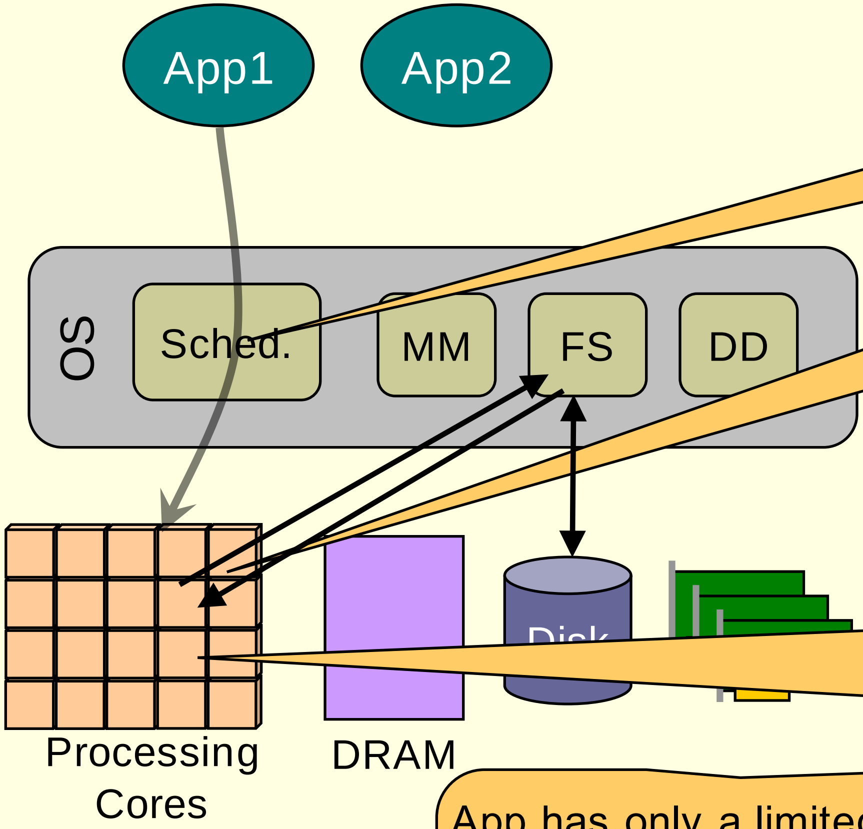


# Organic Operating System (OOS)



- OOS includes learning based ODA loop to optimize resource management
- Observation and control interfaces added to all apps, SW and HW components.
  - Observe temp, heartbeat/performance, miss rates, queue lengths, util. of resources, etc.
  - Control alg., #cores allocated, cache config, scheduler policy, affinity, freq., precision
- Application communicates goals and options to OOS
- OOS uses component perf. models to decide how best to meet goals given global system

# Operation of a Traditional Computer System



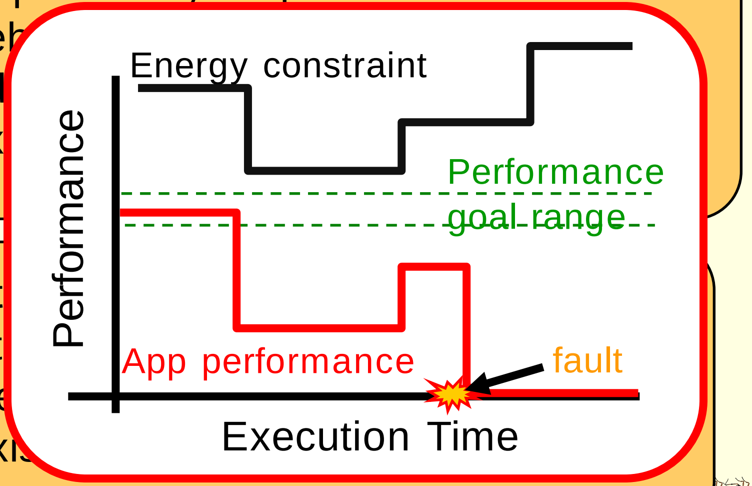
OS schedules an app to run on a core and then transfers control

App runs freely and makes calls to OS services as needed

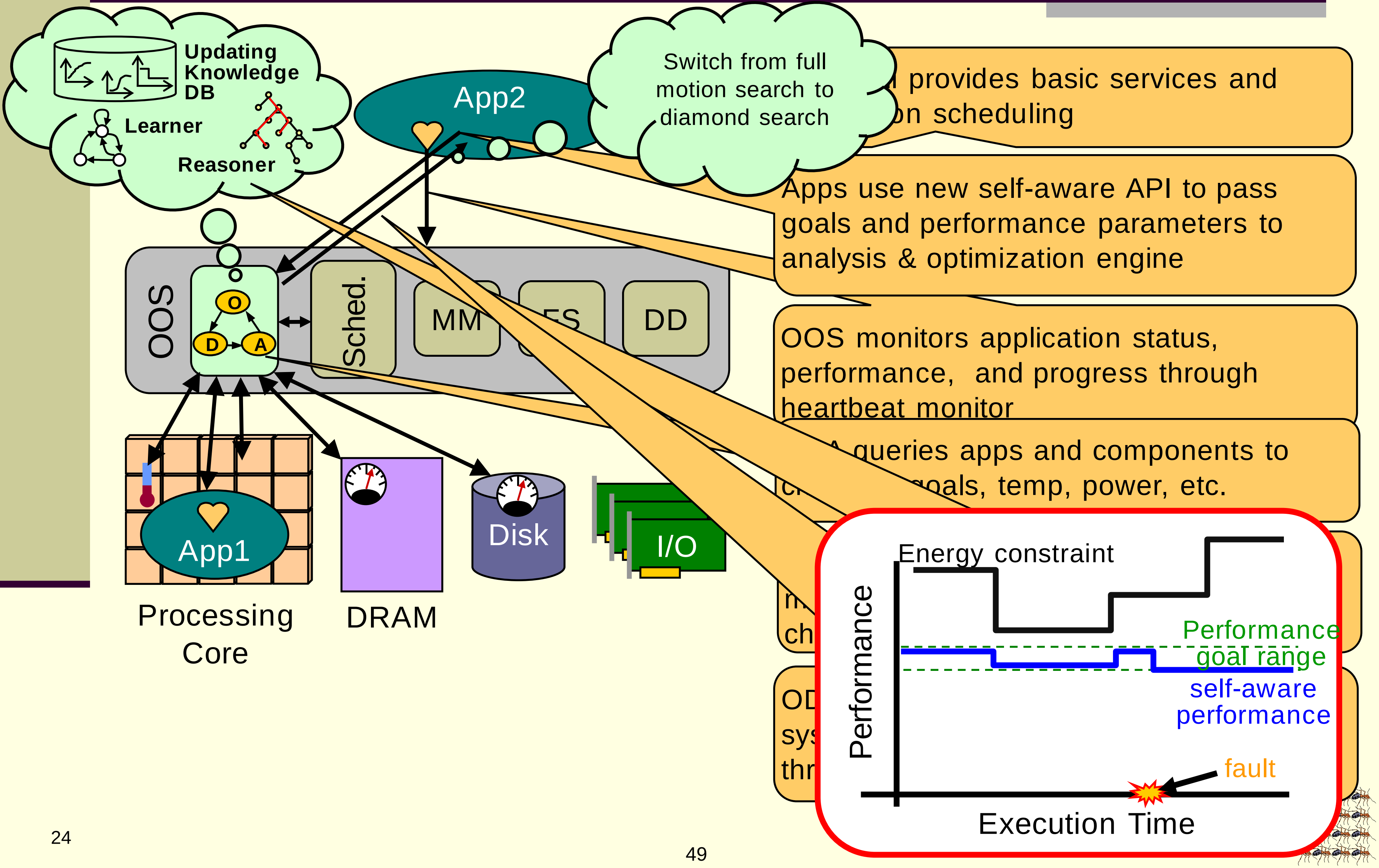
OS only gets involved when app requests it, or to switch to new app

App is solely responsible for its
beh
ad
ex

App has only a limited abilit
hardware and would need t
change of clock freq., cache
this support does not yet exi



# Operation of a Self-Aware Computer System



l provides basic services and on scheduling

Apps use new self-aware API to pass goals and performance parameters to analysis & optimization engine

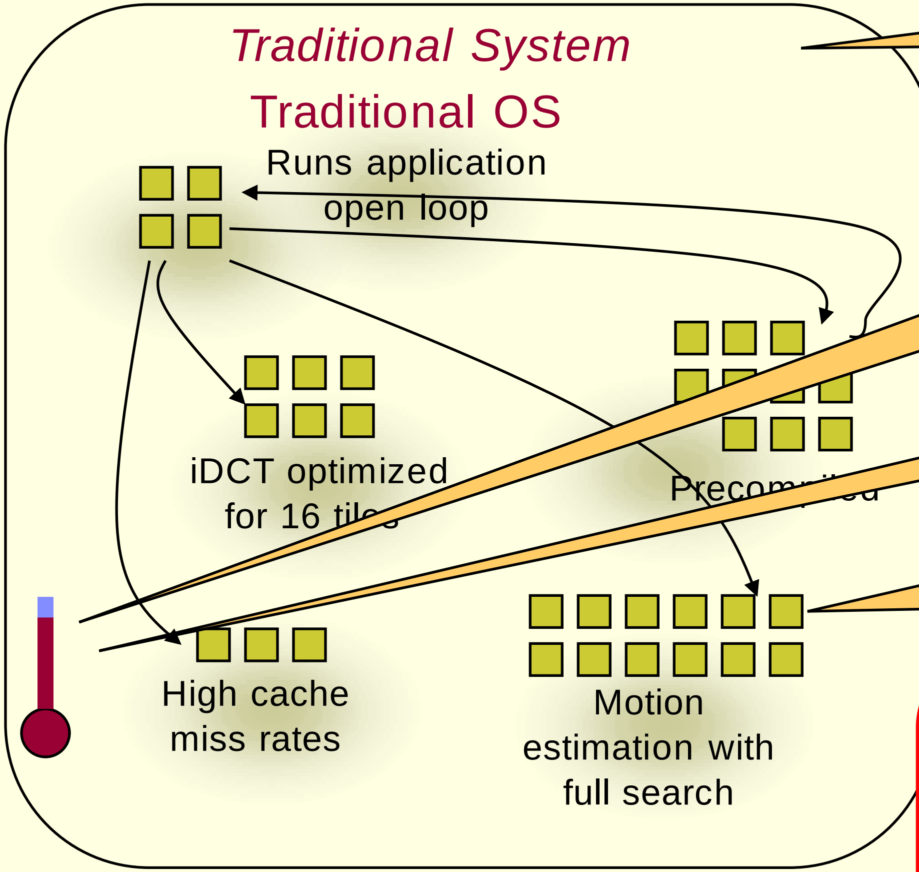
OOS monitors application status, performance, and progress through heartbeat monitor

A queries apps and components to c goals, temp, power, etc.

m ch

OD sys thr

# Example 1: Remote Briefing on Traditional Computing System

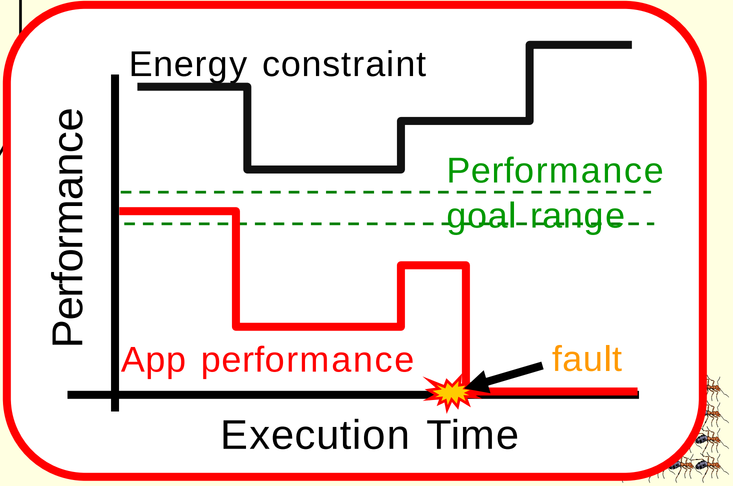

*Traditional System*
Traditional OS
Runs application open loop
iDCT optimized for 16 tiles
Precompiled
High cache miss rates
Motion estimation with full search

Rate starts on target at 30 fps

Temperature rises to 150°C, and battery also drains

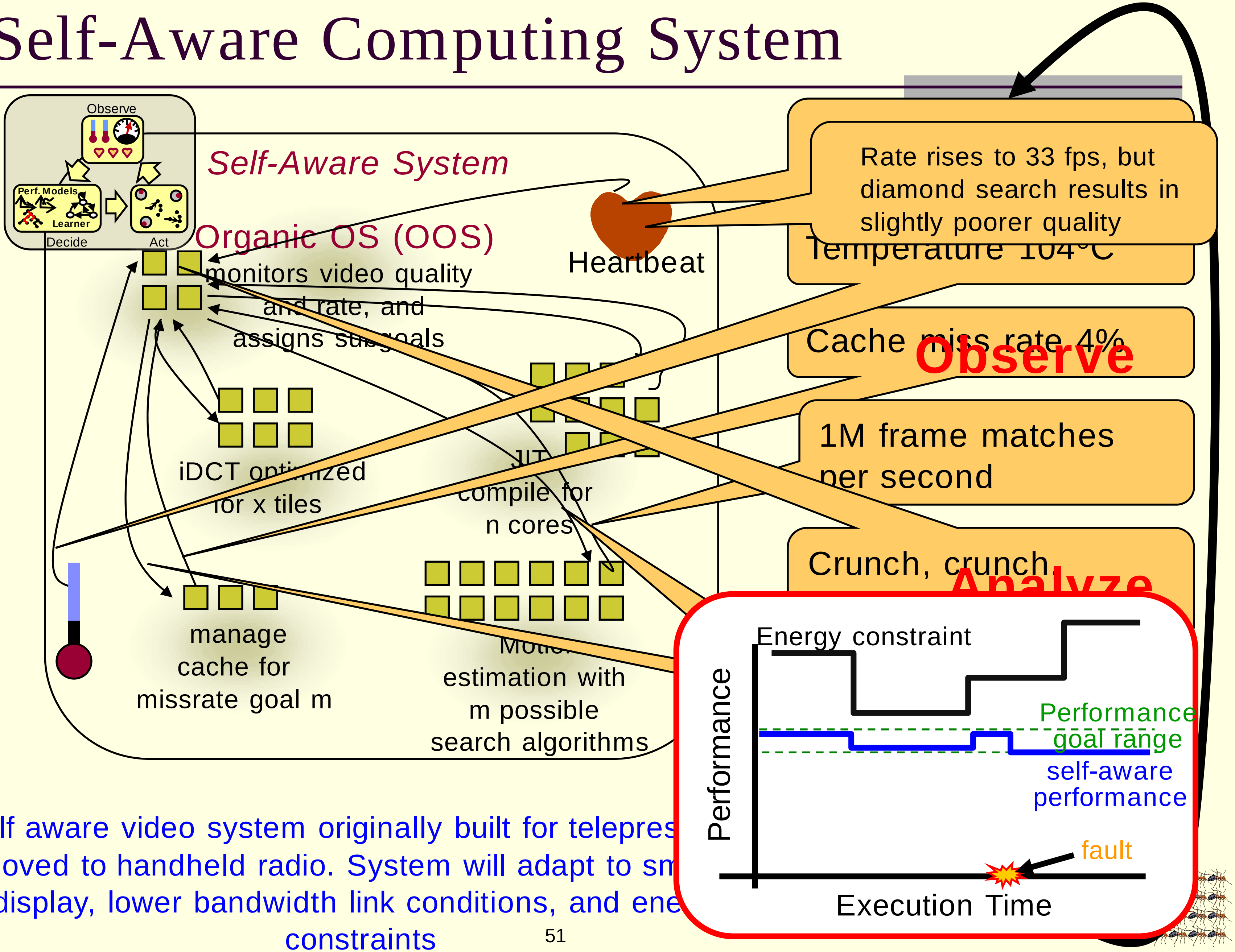
Clock slowed due to high temperature and low energy

Rate falls to 3 fps, video becomes jerky and unusable, and batteries run out in minutes



Video system originally tuned for telepresence using 256 cores, moved to handheld radio with 32 cores. System misses real time goals, batteries drain in minutes

# Example 1: Remote Briefing on Self-Aware Computing System



Rate rises to 33 fps, but diamond search results in slightly poorer quality

Temperature 104°C

Cache miss rate 4%

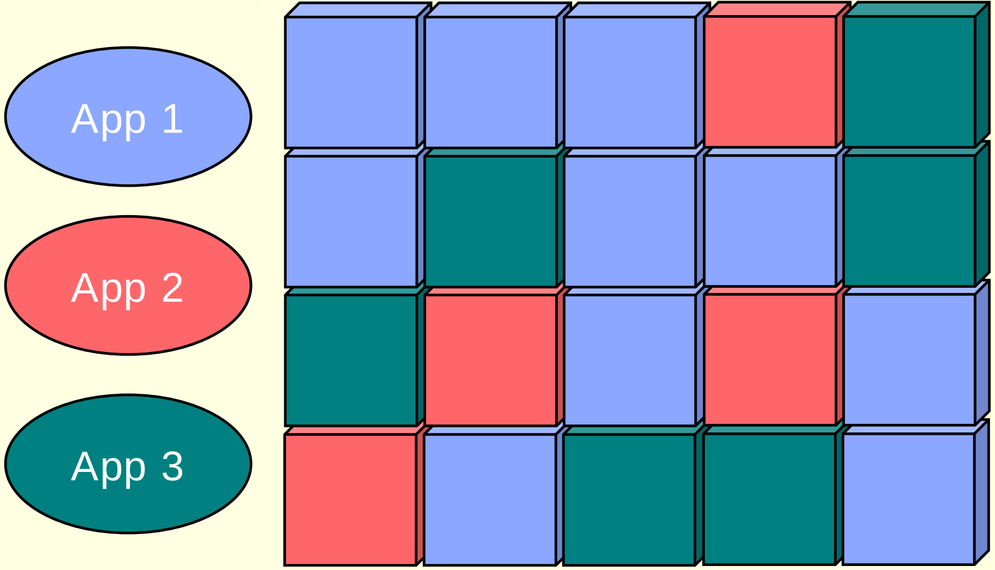
1M frame matches per second

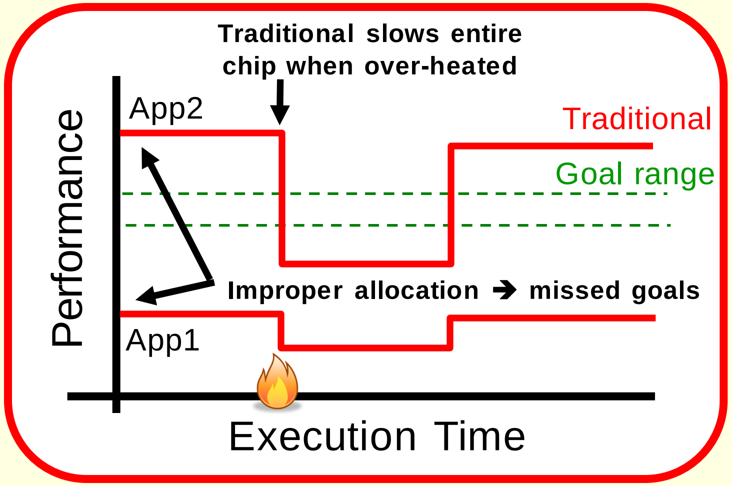
Crunch, crunch

Self aware video system originally built for telepres
moved to handheld radio. System will adapt to sm
display, lower bandwidth link conditions, and ene
constraints

# Example 2: Process Scheduler in Traditional OS

- Scheduler multiplexes a set of applications onto a set of cores
- Applications have no affinity for particular cores
  - Threads just get the next available core and can move randomly
- Apps/threads are swapped at fixed intervals
  - Not sensitive to application’s requirements
- No communication about the amount of parallelism available/needed
  - App creates some number of threads (maybe too many or too few) and throws them over the fence to the scheduler

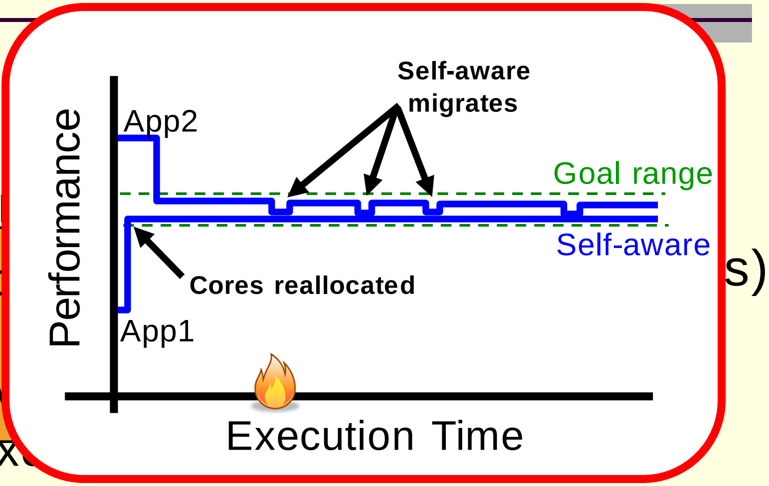


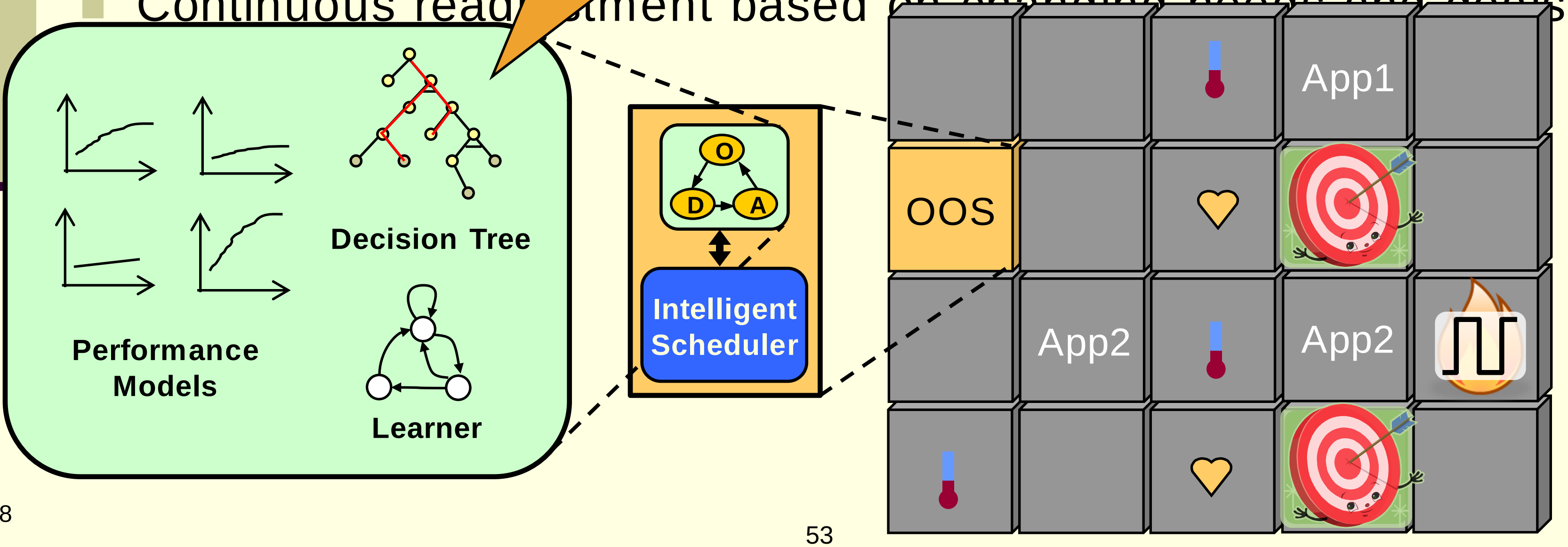


# Example 2: Intelligent Process Scheduler in an OOS

- Scheduler uses spatial information applications
- Applications communicate goals a
- OOS monitors s)
- Scheduler rea
  - Allocate mo
  - Remove cores fro ions ex
  - Reduce clock fre on hot and/or lightly loaded cores
  - Migrate apps to ve communication locality or reduce hotspots
- Continuous readi stment based on changing needs and goals

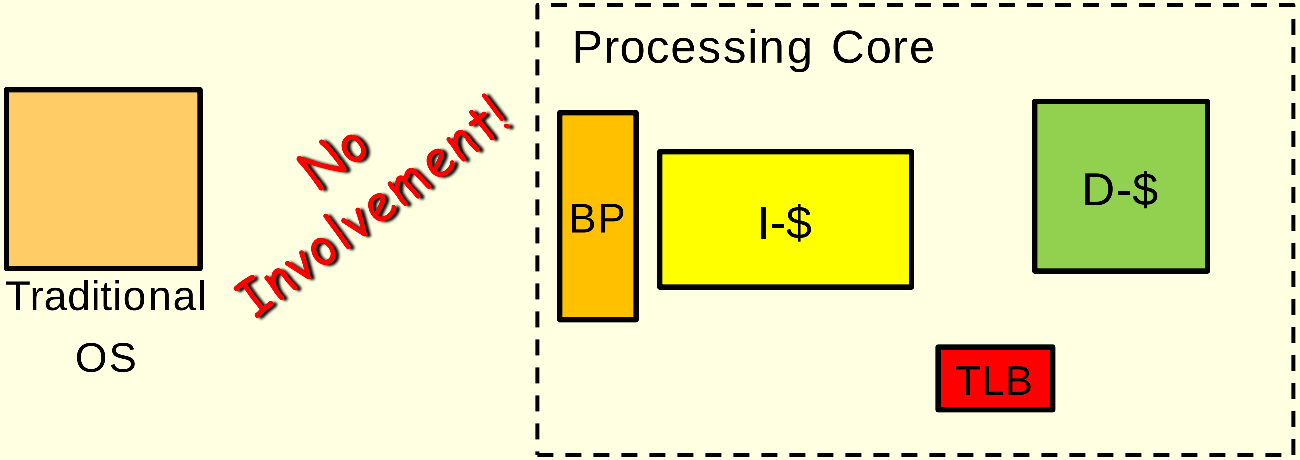
Need to develop mod algorithms to decide take, *e.g.* migrate





# Example 3: Traditional HW Fixed-Size Local Memories

- Current processors have several separate physical memories in each core
  - Instruction and Data Caches
  - Branch Prediction history table
  - Translation Lookaside Buffer (virtual memory mapping cache)
- Fixed allocation of space selected by processor designer
- Fixed-function, managed by hardware --- no opportunity for operating system to reconfigure or optimize behavior



# Example 3: Self-Aware HW Reconfigurable Memory Pool

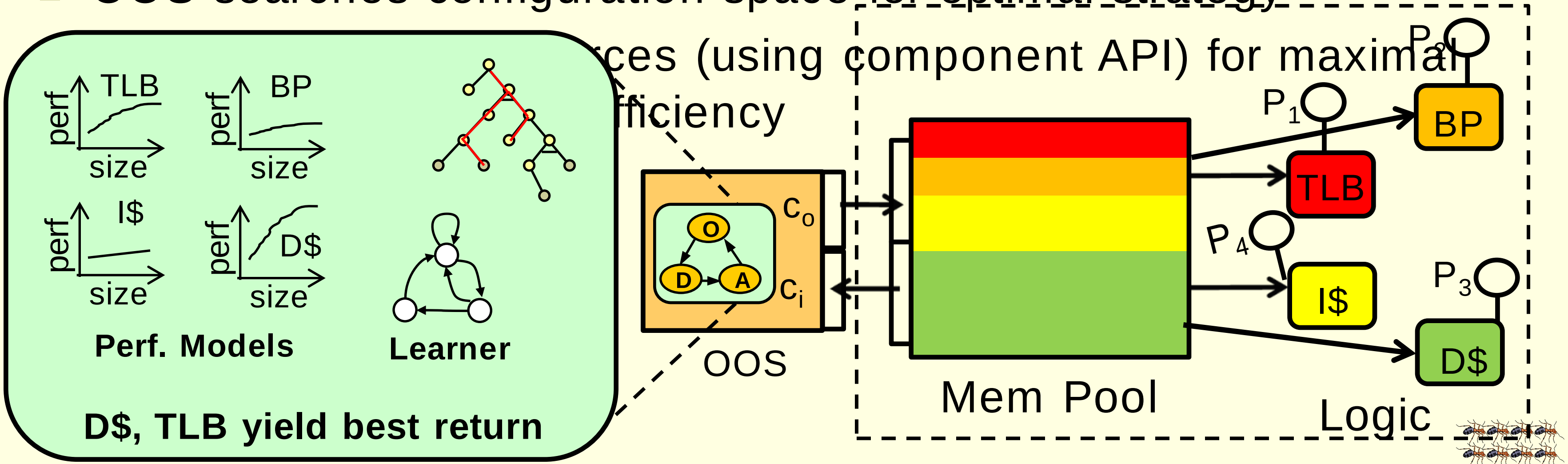
Challenge: Our field needs to develop analytical and empirical models for each component

- Each core contains a pool of lo… d to instruction… …tor, …

…fast methods …onstrained …problem

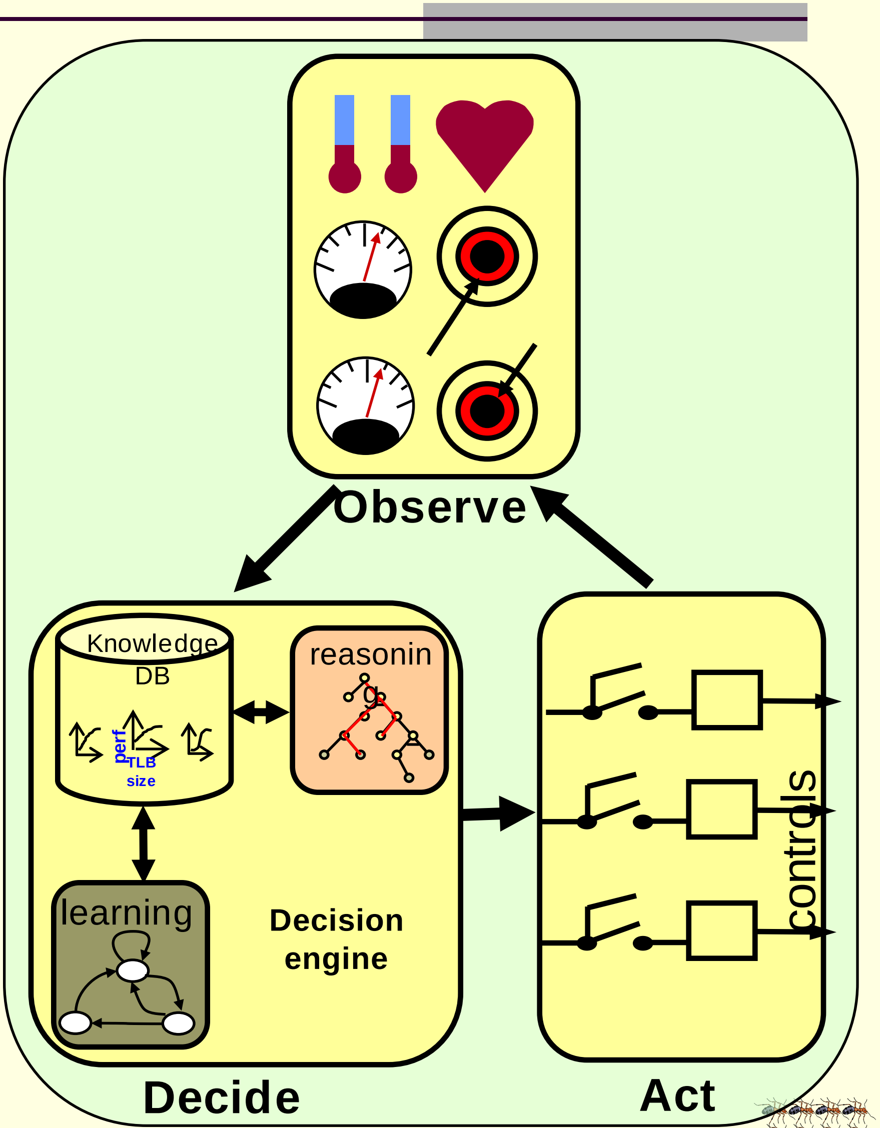
Challenge: Need to develop configurable software and hardware components

- … via pro… $P_i$
- …els for each component
- OOS composes c…mponent result… into global performance model
- OOS searches configuration space for optimal strategy
- …ces (using component API) for maximal …fficiency

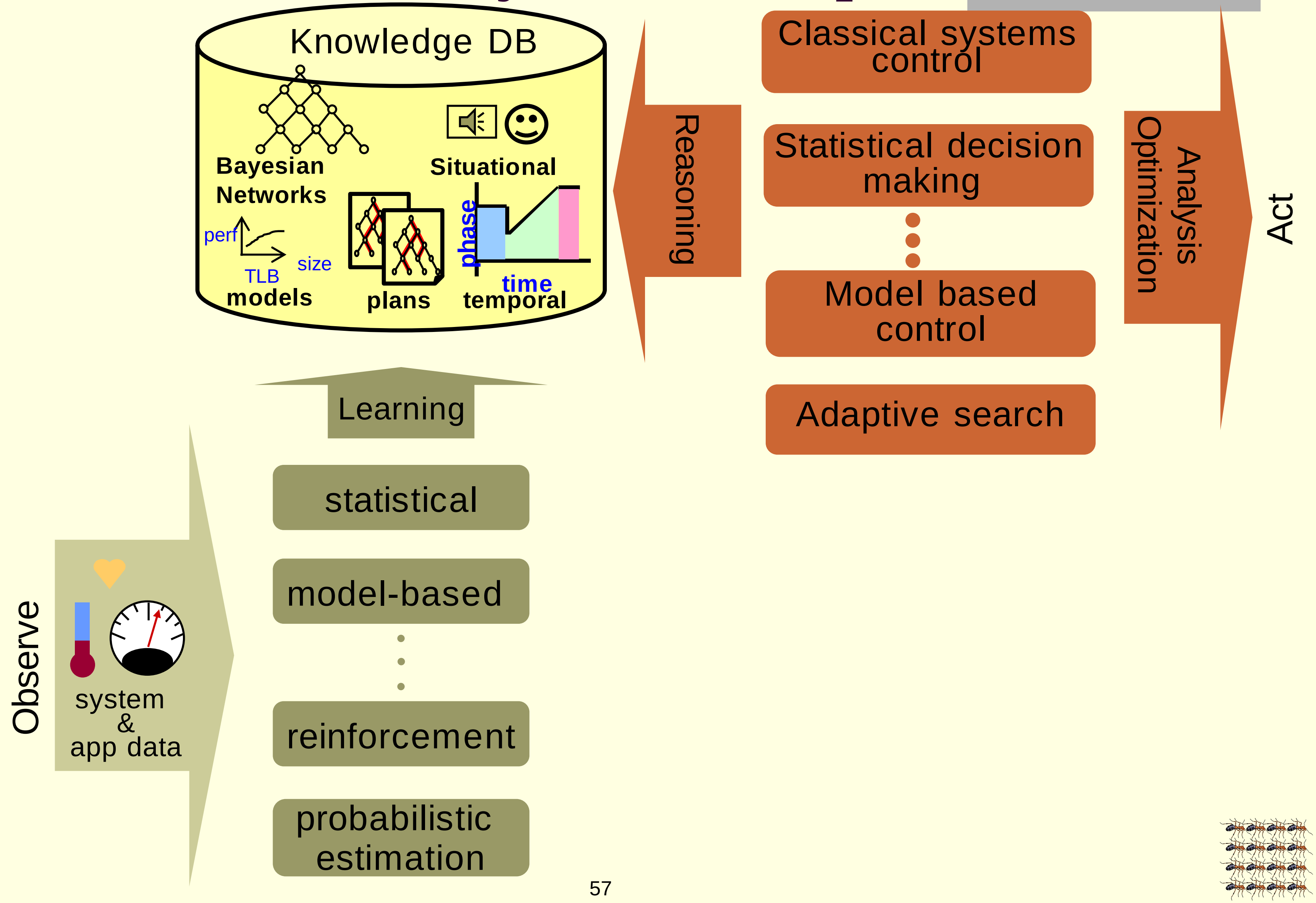


# Intelligent ODA Loop

- ODA loop in the operating system involves observing the situation (O), decision making (D), and taking action (A)
- Decision making (D) is a key component involving learning, knowledge acquisition, and reasoning
  - Learning algorithms update knowledge DB with system and application status
  - Knowledge examples
    - Situational; Eg. system alarms
    - Temporal patterns; Eg. System phases
    - Experience-based; Eg. cases, outcomes
    - Performance-based; Eg. System and application models
    - Operational; How to do it
    - Remedial/diagnostic; How to fix it
  - Reasoning algorithms reference knowledge DB to perform analysis and optimization



# Decision Engine: Performs Analysis and Optimization



Knowledge DB

Bayesian Networks

Situational

models

plans

temporal

Reasoning

Classical systems control

Statistical decision making

Model based control

Adaptive search

Analysis

Optimization

Act

Learning

statistical

model-based

reinforcement

probabilistic estimation

Observe

system & app data

# Key Innovations

- Self-aware OS (not programmer) handles the complexity
  - Programmer specifies goals, OOS figures out how to meet them
  - System is dynamic and adaptable to changing conditions or requirements
  - Same program will work on many different systems
  - System detects and handles errors as needed
    - Degree of reliability can be specified as a goal
  - Example: Fault-tolerance
    - User simply requests a required degree of reliability
    - OOS enables error detection routines and hardware
    - OOS rolls back affected core after soft error, or routes around faults

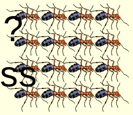
- System-level goals and constraints
  - Current systems use individual models and constraints
  - Current systems must overprovision resources in all dimensions
  - Self-aware system ensures that goals are met without exceeding global constraints (e.g. maximum power, temperature)

# Technical Challenges

- Self aware operating system
  - What are good learning and optimization algorithms for the OS to use when selecting configurations? Competing technologies include
    - Learning engines
    - Neural networks
    - Genetic models
    - Classical control theory
    - Stochastic control
  - How to create predictive performance and related models for the OS to be able to assess the expected impact of its actions
  - Performance and energy cost tradeoffs in the OOS itself
  - How can we synchronize the timescale of learning and speed of adaptation with system and application shifts?

- Self aware components and applications
  - How do we build reconfigurable (controllable), introspective (observable) components?
  - What are the miminal changes required in current software (e.g., apps, paging systems, schedulers, device drivers, memory allocators, data structure libraries), and hardware components (e.g., caches, processors, networks, I/O devices, disks, coherence engines, DRAM) to enable self aware operation
  - How do we create simplified performance, reliability, and energy models of these components for the OS to use
    - E.g. How will changing cache associativity affect miss rate, power, temperature

- Self aware APIs
  - What are the right APIs between the operating system and self-aware components?
  - What information do applications communicate to the OOS for it to be able to assess the performance and other implications of configuration changes

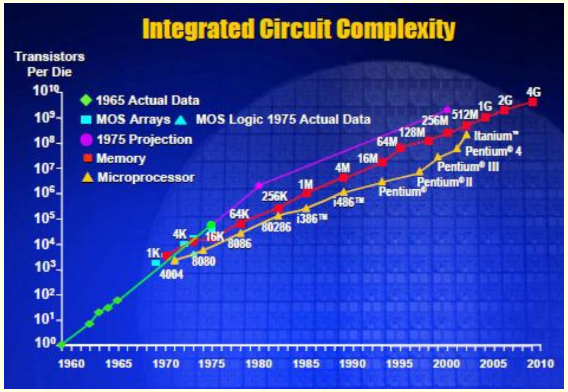


# Outline

- Introduction
- Vision and Goals
- Limitations of Today's Systems
- How to Build a Self-Aware System
- Why Now?
- Potential Impact
- Program Timeline

# Why Now?

- Resources have changed dramatically over the years
  - Quantity of transistors, memory, have increased exponentially
  - Speed of components has increased by orders of magnitude
  - Level of integration has increased dramatically
- We continue to apply outdated approaches to using these resources
  - Key computer science abstractions have not changed since the 1960’s
  - Operating systems, languages, etc. we use today were designed for a different era
  - For example, OS still time multiplexes a single core between OS and user app – applies to the era when compute resources were expensive
- It’s time for a fresh approach to the way systems are designed and used
  - Leverage the new balance of resources to improve performance, utilization, reliability and programmability
  - We will first address the OS, followed by architectural support



# The New Balance of Resources

Advancing VLSI technology and multicore architectures have created a disruptive cost inversion!

**BEFORE**

- Resources (cores) were expensive
- Energy was cheap
- Sequential programming was easy
- Devices were fairly reliable

**NOW**

- Transistors and cores are free
- Energy is a first-order constraint
- Programming is hard
- Hard and soft error rates are increasing

Massive multicore is both the problem *and* the enabler of the solution

Self-aware computing makes a new set of tradeoffs

**Trade these…**

- Transistors
- Cores
- On-chip bandwidth

E.g., throw transistors at the problem of creating extensive counters

**for these…**

- Energy
- Time to completion
- Programmer effort
- Reliability
- Off-chip bandwidth

# Building on Key New Technologies

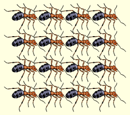
- Previous projects have created the key mechanisms
  - DARPA Polymorphic Computing Architectures
    - Polymorphic, adaptable hardware
    - But, manually reconfigured/adapted
    - Did not address faults
  - DARPA Power Aware Computing and Communications
    - Energy monitoring mechanisms
  - Recent advancements in learning and reasoning algorithms
  - Continuous optimization technologies, but in limited domains
    - IBM continuous program optimization
      - Framework to collect system data
      - Dynamic adjustment of memory page size
      - Remembers Java JIT optimization heuristics between runs
    - Auto tuners, FFTW, STAPL, Atlas, OTL
    - Virtual architecture reconfiguration [CGO 2006]
      - Used in architectural emulation
      - Inspect current architecture and instruction stream
      - Change config. to suit program's needs (e.g., number of tiles allocated to trace caching)
- Self-Aware Computing will create the high-level operating system necessary to automatically use and coordinate these primitive mechanisms and apply them to build broadly applicable general-purpose systems

# Outline

- Introduction
- Vision and Goals
- Limitations of Today’s Systems
- How to Build a Self-Aware System
- Why Now?
- Potential Impact
- Program Timeline

# Impact of Self-Aware Computing Program

- Make complex systems easy to program
  - Utilize massive resources efficiently
  - Hide complexity from user
- Translate massive silicon resources into performance benefits
- Increase portability of software to different systems
  - Decreases development costs
- Enable *dynamic* adaptation to changing mission requirements or operating conditions
  - Currently very difficult to statically pre-program all possible variations
- Reduces verification effort and provides a safety net
  - System ensures correct behavior by enforcing goals and constraints
  - Test for known variables, system handles unknowns
- Maximize energy efficiency in highly dynamic situations
  - System expends only the energy necessary for current conditions



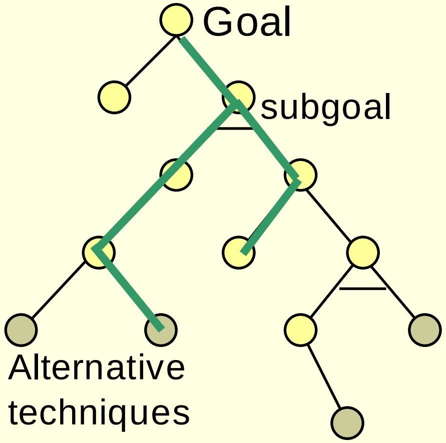
# Impact of Self Aware Computing Program

Achieve orders-of-magnitude improvements in each of four key computing criteria

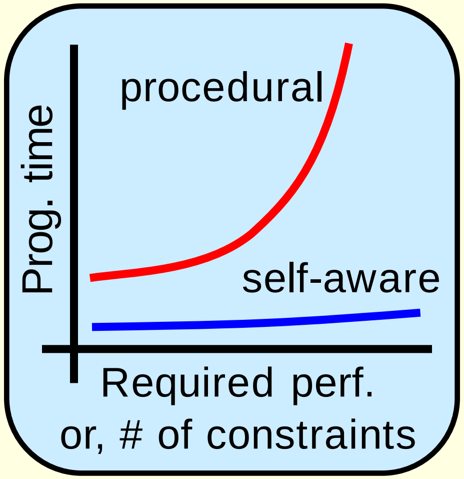
1. Programming Effort
   - 10X decrease in average time to implement solution to achieve a given level of  performance under a specific set of constraints
   - 100X increase in performance of quick implementation
2. Reliability
   - 10X-100X increase in error-free operation time given faults
3. Energy
   - 10X-100X decrease in energy as system optimizes itself
4. Performance
   - 10X increase in performance while meeting constraints from initial run to steady state

# 1. Programming Effort Example

Users/Applications focus on goals, not how to manipulate the minutiae of complex manycores to accomplish goals

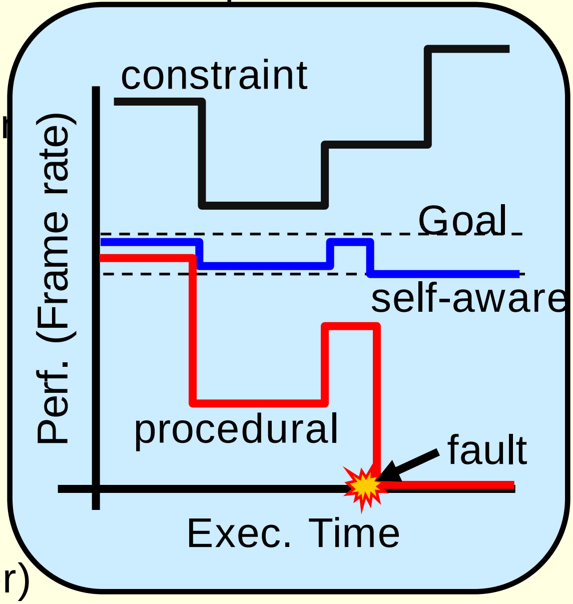


- Application is no longer a single sequence of procedural code
- Application specifies goals and provides library of alternative techniques
  - OOS runtime may supply libraries of techniques for common ops. E.g., a data repository with alternatives including hash-table vs. red-black tree, list vs. heap, row-major vs. column-major order, sparse vs. dense matrix
- Application passes goals and technique properties to OOS runtime through self aware API
- OOS creates tree of subgoals and develops a “plan” to achieve goals (a plan is a set of techniques)
  - Leverage machine learning algorithms
  - Plan is dynamically adjusted based on changing conditions/mission requirements
  - Programmer does not need to understand interaction of different tradeoffs
- OOS may create different plans for the same goals on different machines or runtime conditions (program portability)
- OOS configures system SW and HW to best suit application
  - E.g., OOS runtime observes usage patterns and reorganizes data structures to optimize common ops



# 2. Reliability Examples

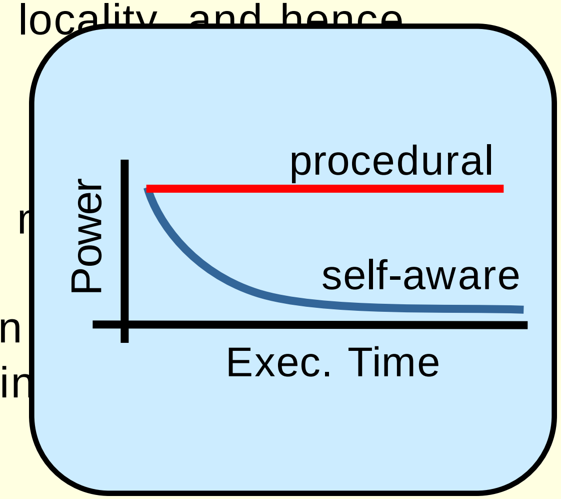
- Future exascale processors containing 100's of billions of transistors will experience frequent errors
- Using the current procedural approach, applications will fail to complete due to corruption
- OOS can automatically monitor machine and applicatio actions to ensure reliable execution
- Handling soft errors (e.g., bit-flips due to cosmic rays)
  - OOS duplicates computation and compares execution UIUC]
  - Hardware introspection mechanisms to observe errors
  - Roll back only the affected core and continue
  - Tune checkpointing interval, datasets
- Hard faults (e.g., broken functional unit or on-chip router)
  - OOS periodically self tests and characterizes machine components
  - If a component is found to be broken, dynamically reorganizes computation to avoid failed hardware
    - Alter on-chip routing tables
    - Dynamically rewrite code to avoid functional unit
    - Use JIT to recompile for a different number/configuration of cores



# 3. Energy Examples

- Continually optimizing energy by adjusting core parameters
  - Organic OS dynamically monitors and recompiles application for energy
  - Target varying numbers of cores, shutting the others off
  - Adjust voltage and frequency of each core to maintain minimum required throughput
  - Migrates processes to increase communication locality, and hence energy efficiency

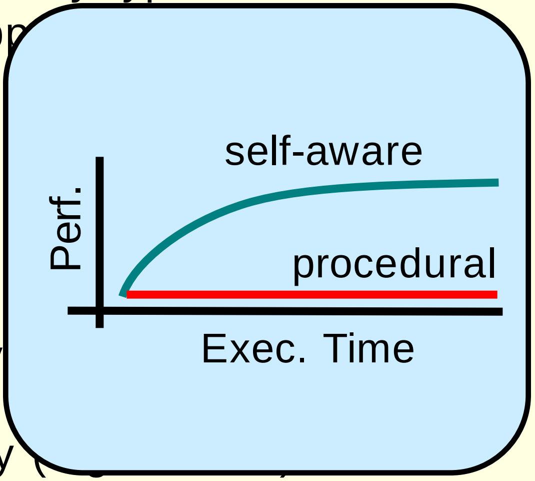
- Optimizing energy by reconfiguring memories
  - OOS monitors cache usage and adjusts size to r
  - Extra cache is turned off to save energy
  - Alter cache hash functions to optimize hit-rate in
  - 70% of server power in external DRAM [Hetherin set based memory tuning



- Optimizing I/O energy
  - Dynamically adjust I/O voltage and frequency to meet bandwidth requirements [Balamurugan, VLSI Circuits 07]
  - Use predictive bus-coding [Wen, HPCA04] to reduce on-chip communication energy

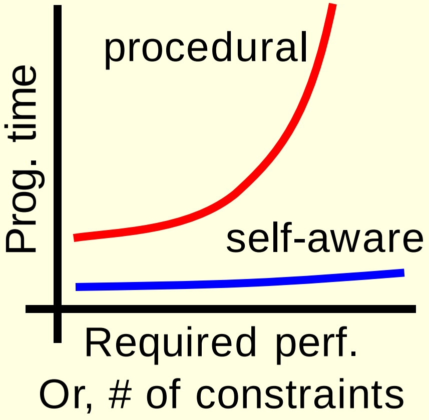
# 4. Performance Examples

- Key concept: Use introspection and dynamic reconfiguration by Organic OS to optimize software/hardware for a specific application
- Application-aware dynamic cache memory pool
  - OOS observes app behavior and miss rates of each type of memory
  - Has models of performance vs. size of each memory type
  - Predicts and establishes best config for given app
- Adaptive Page Allocation
  - OOS monitors memory controllers, queues, and
  - Remaps pages to balance load on all controllers fair to all applications
- Approximate computation
  - OOS receives programmer/app specified energy quality requirements
  - Observes dynamic energy use and output quality
  - Through self aware API suggests algorithmic policies to app
  - Prune unnecessary computation and adjusts #cores used
  - OOS controls JIT compiler to recompile app for different #cores
  - Uses SIMD hardware to perform multiple smaller computations at once, for energy savings, but at a loss in accuracy

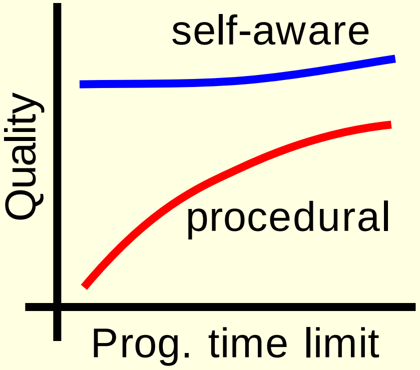


# Metrics for Evaluation

- **Programming Effort:** Time to required solution
  - Addresses: programming effort
  - Create a set of “benchmark” problems with specific goals (e.g. performance) and constraints (e.g. energy)
  - Measure the average time it takes programmers to implement the benchmarks
  - Expected 10X reduction vs. traditional system
  - Why?: OOS frees the programmer from worrying about constraint satisfaction and helps tune the architecture and application automatically

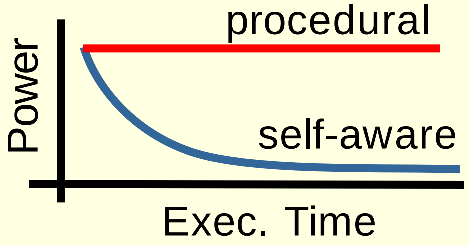


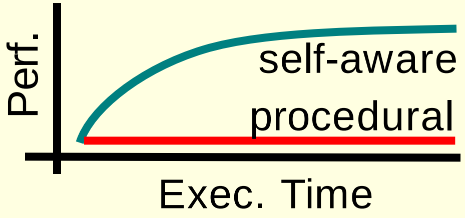
- **Solution Quality:** Performance of quick implementation
  - Addresses: performance, energy, resiliency, prog. effort
  - Specify benchmark problems and a programming time limit
  - Measure performance, energy efficiency, resiliency of resulting solutions
  - Resiliency = length of error-free operation with injected faults
  - Expected 10X to 100X improvement vs. traditional system
  - Why?: OOS handles performance tuning and fault tolerance



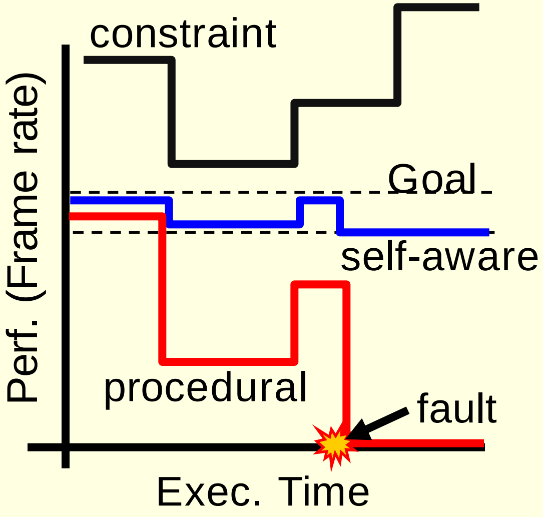
# Metrics for Evaluation (cont.)

- **Continuous Optimization:** Performance/Power improvement between initial run and steady state
  - Addresses: adaptability, performance, energy
  - Run benchmarks for a long period of time
  - Performance/power should improve over time
  - Expected 10X-100X improvement between initial run and steady state
  - Why?: OOS observes system behavior and optimizes continuously





- **Performance Stability:** Adaptation to changing conditions as constraints and input conditions change
  - Addresses: adaptability, resiliency
  - Supply a set of applications and a profile of goals and constraints that change over time
  - System must maintain goals and constraints, free variables may change
  - Example: Video encoding
    - Energy constraint
    - Frame rate and resolution goals
    - Video quality may vary
  - Why?: OOS continually monitors goals and adjusts system to compensate for dynamic events/changes



# Outline

- Introduction
- Vision and Goals
- Limitations of Today’s Systems
- How to Build a Self-Aware System
- Why Now?
- Potential Impact
- Program Timeline

# 3-Phase Program

- Phase 1 (12 months)
  - High-level OOS design
  - Early proof of concept simulation studies
  - Identify key learning, introspection and adaptation mechanisms and algorithms
- Phase 2 (24 months)
  - Complete simulation studies
  - Build prototype OOS using existing hardware with selected self aware modules
- Phase 3 (24 months)
  - Modify hardware components to make them self aware
  - Integrate with OOS
  - Build complete system
  - Evaluate

# Backup slides

# Background: An Operating System

*Software layer responsible for managing a computer system’s resources and activities*

- Abstracts away details of hardware to simplify application programming. Does so by providing abstract APIs (e.g., to print) and device specific drivers (e.g., printer driver)
- Coordinates sharing of resources between multiple applications
  - Memory resources: OS memory protection, virtual memory
  - CPU resources: OS Scheduler decides which application gets the CPU next and for how long
  - I/O device resources: Device drivers arbitrate use of hardware devices
- Disk access and file systems
  - Provides abstract concept of “files” to applications, handles all details of how they are actually stored
- Network stack for communicating with other machines
- User interface
  - Not necessarily part of all operating systems (e.g. UNIX)